

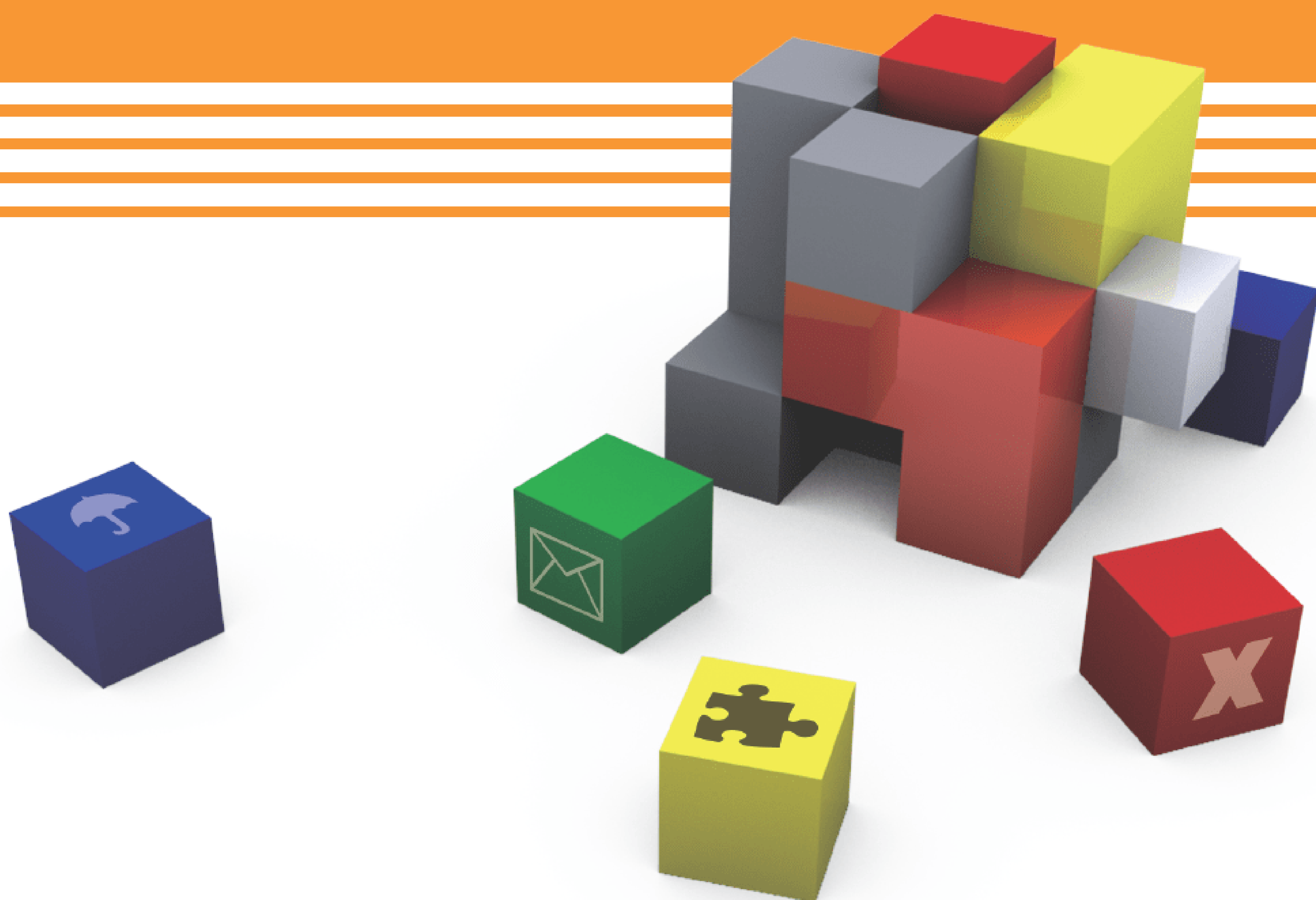
"十二五"普通高等教育本科国家级规划教材

高等学校数据结构课程系列教材

（第5版）

数据结构教程学习指导

李春葆 主编



- 高度概括本章知识结构图、基本知识点和要点归纳。
- 提供主教材中练习题的参考答案。
- 提供各种题型的练习题及参考答案。
- 提供不同类型的考试试卷及参考答案。

清华大学出版社

“十二五”普通高等教育本科国家级规划教材
高等学校数据结构课程系列教材

数据结构教程(第5版)
学习指导

李春葆　主编
尹为民　蒋晶珏　喻丹丹　蒋林　编著

清华大学出版社
北京

内 容 简 介

本书是《数据结构教程(第5版)》(李春葆等编著,清华大学出版社出版)的配套学习指导书。两书章节一一对应,内容包括绪论、线性表、栈和队列、串、递归、数组和广义表、树和二叉树、图、查找、内排序、外排序和文件。各章中除给出本章练习题的参考答案以外还总结了本章的知识体系结构,并补充了大量的练习题且予以解析,因此自成一体,可以脱离主教材单独使用。

本书适合高等院校计算机和相关专业的本科生及研究生使用。

图书在版编目(CIP)数据

数据结构教程(第5版)学习指导/李春葆主编.—北京:清华大学出版社,2017
(高等学校数据结构课程系列教材)
ISBN 978-7-302-45587-5

Ⅰ.①数… Ⅱ.①李… Ⅲ.①数据结构-教学参考资料 Ⅳ.①TP311.12

中国版本图书馆CIP数据核字(2016)第283889号

责任编辑:魏江江 王冰飞
封面设计:
责任校对:时翠兰
责任印制:

出版发行:清华大学出版社
网　址:http://www.tup.com.cn,http://www.wqbook.com
地　址:北京清华大学学研大厦A座　**邮　编:**100084
社 总 机:010-62770175　**邮　购:**010-62786544
投稿与读者服务:010-62776969,c-service@tup.tsinghua.edu.cn
质量反馈:010-62772015,zhiliang@tup.tsinghua.edu.cn
课件下载:http://www.tup.com.cn,010-62795954
印 刷 者:
装 订 者:
经　销:全国新华书店
开　本:185mm×260mm　**印　张:**22.75　**字　数:**554千字
版　次:2017年7月第1版　**印　次:**2017年7月第1次印刷
印　数:1～ 000
定　价: .00元

产品编号:072425-01

前　言　Preface

本书是《数据结构教程(第5版)》(清华大学出版社,以下简称为《教程》)的配套学习指导书。全书分为12章,第1章为绪论;第2章为线性表;第3章为栈和队列;第4章为串;第5章为递归;第6章为数组和广义表;第7章为树和二叉树;第8章为图;第9章为查找;第10章为内排序;第11章为外排序;第12章为文件。本书各章次与《教程》的章次相对应。附录A给出了两份本科生期末考试试题及参考答案,附录B给出了两份研究生入学考试(单考)数据结构部分试题及参考答案,附录C给出了两份全国计算机学科专业考研题数据结构部分试题及参考答案。

每章包括以下内容。

- 本章知识体系:高度概括本章知识结构图、基本知识点和要点归纳。
- 教材中的练习题及参考答案:给出了《教程》中对应章节练习题的参考答案。
- 补充练习题及参考答案:列出了大量相关的练习题,并按单项选择题、填空题、判断题、简答题和算法分析题或算法设计题分类,同时给出了这些题目的参考答案。其中许多题目是多年来全国各高校计算机专业的数据结构考研题。

书中列出了全部的练习题题目,因此自成一体,可以脱离《教程》单独使用。

由于水平所限,尽管编者不遗余力,仍可能存在错误和不足之处,敬请教师和同学们批评指正。

编　者

2017年1月

目 录 Contents

第1章 绪论

1.1 本章知识体系



1. 知识结构图

本章的知识结构如图 1.1 所示。



图 1.1　第 1 章知识结构图

2. 基本知识点

（1）数据的逻辑结构、存储结构和数据运算三方面的概念及相互关系。

（2）采用抽象数据类型描述求解问题。

（3）算法描述中的输出型参数描述方法。

（4）算法的时间和空间复杂度分析，特别是递归算法的时间和空间复杂度分析。

（5）如何设计“好”的算法。

3. 要点归纳

（1）数据结构是相互之间存在一种或多种特定关系的数据元素的集合。数据是由数据元素组成的，数据元素可以由若干个数据项组成，数据元素是数据的基本单位，数据项是数据的最小单位。

（2）数据结构一般包括数据逻辑结构、数据存储结构和数据运算 3 个方面。数据运算

分为抽象运算(运算功能描述)和运算实现两个层次。

(3) 数据的逻辑结构分为集合、线性结构、树形结构和图形结构,树形结构和图形结构统称为非线性结构。

(4) 数据的存储结构分为顺序存储结构、链式存储结构、索引存储结构和哈希(散列)存储结构。

(5) 在设计数据的存储结构时既要存储逻辑结构的每个元素值,又要存储元素之间的逻辑关系。同一逻辑结构可以设计相对应的多个存储结构。

(6) 描述一个求解问题的抽象数据类型由数据逻辑结构和抽象运算两部分组成。

(7) 算法是对特定问题求解步骤的一种描述,它是指令的有限序列。运算实现通过算法来表示。

(8) 算法具有有穷性、确定性、可行性、输入和输出 5 个重要特征。

(9) 算法满足有穷性,程序不一定满足有穷性。算法可以用计算机程序来描述,但并不是说任何算法必须用程序来描述。

(10) 在用 C/C++语言描述算法时,通常算法采用 C/C++函数的形式来描述,复杂算法可能需要多个函数来表示。

(11) 在设计一个算法时先要弄清哪些是输入(已知条件)、哪些是输出(求解结果),通常将输入参数设计成非引用型形参,将输出参数设计成引用型形参。在有些情况下,算法求解结果可以用函数返回值表示。

(12) 对于算法的输入通常需要判断其有效性,当输入有效并正确执行时返回 true(真),否则返回 false(假)。

(13) 算法分析包括时间复杂度和空间复杂度分析,其目的是分析算法的效率以求改进,所以通常采用事前估算法,而不是进行算法绝对执行时间的比较。

(14) 在分析算法的时间复杂度时通常选取算法中的基本运算,求出其频度,取最高阶并置序数为 1 作为该算法的时间复杂度。

(15) 递归算法的时间复杂度分析方法是先由递归算法推导出执行时间的递推式,然后计算 $T(n)$,再用 O 表示。

(16) 递归算法的空间复杂度分析方法是先由递归算法推导出占用空间的递推式,然后计算 $S(n)$,再用 O 表示。

(17) 通常算法是建立在数据存储结构之上的,设计好的存储结构可以提高算法的效率。

(18) 求解问题的一般步骤是建立其抽象数据类型,针对运算的实现设计出合理的存储结构,在此基础上设计出尽可能高效的算法。

1.2 教材中的练习题及参考答案



1. 简述数据与数据元素的关系与区别。

答: 凡是能被计算机存储、加工的对象统称为数据,数据是一个集合。数据元素是数据的基本单位,是数据的个体。数据元素与数据之间的关系是元素与集合之间的关系。

2. 采用二元组表示的数据逻辑结构 S=<D,R>,其中 $D=\{a,b,\cdots,i\}$,$R=\{r\}$,$r=\{<a,b>,<a,c>,<c,d>,<c,f>,<f,h>,<d,e>,<f,g>,<h,i>\}$,问:关系 r 是什么类型的逻辑结构?哪些结点是开始结点,哪些结点是终端结点?

答:该逻辑结构为树形结构,其中 a 结点没有前驱结点,它是开始结点,b、e、i 和 g 、结点没有后继结点,它们都是终端结点。

3. 简述数据逻辑结构与存储结构的关系。

答:在数据结构中,逻辑结构与计算机无关,存储结构是数据元素之间的逻辑关系在计算机中的表示。存储结构不仅将逻辑结构中的所有数据元素存储到计算机内存中,而且还要在内存中存储各数据元素间的逻辑关系。通常情况下,一种逻辑结构可以有多种存储结构,例如线性结构可以采用顺序存储结构或链式存储结构表示。

4. 简述数据结构中运算描述和运算实现的异同。

答:运算描述是指逻辑结构施加的操作,而运算实现是指一个完成该运算功能的算法。它们的相同点是运算描述和运算实现都能完成对数据的"处理"或某种特定的操作,不同点是运算描述只是描述处理功能,不包括处理步骤和方法,而运算实现的核心是设计处理步骤。

5. 数据结构和数据类型有什么区别?

答:数据结构是相互之间存在一种或多种特定关系的数据元素的集合,一般包括 3 个方面的内容,即数据的逻辑结构、存储结构和数据的运算。数据类型是一个值的集合和定义在这个值集上的一组运算的总称,如 C 语言中的 short int 数据类型是由 −32 768～32 767(16 位机)的整数和+、−、*、/、%等运算符构成的。

6. 在 C/C++ 中提供了引用运算符,简述其在算法描述中的主要作用。

答:在算法设计中,一个算法通常用一个或多个 C/C++ 函数来实现,在 C/C++ 函数之间传递参数时有两种情况,一是从实参到形参的单向值传递;二是实参和形参之间的双向值传递。对形参使用引用运算符即在形参名前加上"&",不仅可以实现实参和形参之间的双向值传递,而且使算法设计简单、明晰。

7. 有以下用 C/C++ 语言描述的算法,说明其功能:

```
void fun(double &y,double x,int n)
{   y = x;
    while (n>1)
    {   y = y * x;
        n--;
    }
}
```

答:本算法的功能是计算 $y=x^n$。

8. 用 C/C++ 语言描述下列算法,并给出算法的时间复杂度。

(1) 求一个 n 阶二维整数数组的所有元素之和。

(2) 对于输入的任意 3 个整数,将它们按从小到大的顺序输出。

(3) 对于输入的任意 n 个整数,输出其中的最大和最小元素。

答：（1）算法如下。

```
int sum(int A[N][N], int n)
{   int i, j, s = 0;
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            s = s + A[i][j];
    return(s);
}
```

本算法的时间复杂度为 $O(n^2)$。

（2）算法如下。

```
void order(int a, int b, int c)
{   if (a > b)
    {   if (b > c)
            printf("%d, %d, %d\n", c, b, a);
        else if (a > c)
            printf("%d, %d, %d\n", b, c, a);
        else
            printf("%d, %d, %d\n", b, a, c);
    }
    else
    {   if (b > c)
        {   if (a > c)
                printf("%d, %d, %d\n", c, a, b);
            else
                printf("%d, %d, %d\n", a, c, b);
        }
        else printf("%d, %d, %d\n", a, b, c);
    }
}
```

本算法的时间复杂度为 $O(1)$。

（3）算法如下。

```
void maxmin(int A[], int n, int &max, int &min)
{   int i;
    min = min = A[0];
    for (i = 1; i < n; i++)
    {   if (A[i]> max)  max = A[i];
        if (A[i]< min)  min = A[i];
    }
}
```

本算法的时间复杂度为 $O(n)$。

9. 设 3 个表示算法频度的函数 f、g 和 h 分别为：

$$f(n) = 100n^3 + n^2 + 1000$$

$$g(n) = 25n^3 + 5000n^2$$
$$h(n) = n^{1.5} + 5000n\log_2 n$$

求它们对应的时间复杂度。

答：$f(n)=100n^3+n^2+1000=O(n^3)$，$g(n)=25n^3+5000n^2=O(n^3)$

当 $n\to\infty$ 时，$\sqrt{n}>\log_2 n$，所以 $h(n)=n^{1.5}+5000n\log_2 n=O(n^{1.5})$。

10. 分析下面程序段中循环语句的执行次数。

```
int j = 0,s = 0,n = 100;
do
{   j = j + 1;
    s = s + 10 * j;
} while (j < n && s < n);
```

答：$j=0$，第 1 次循环 $j=1$，$s=10$。第 2 次循环 $j=2$，$s=30$。第 3 次循环 $j=3$，$s=60$。第 4 次循环 $j=4$，$s=100$。while 条件不再满足。所以其中循环语句的执行次数为 4。

11. 设 n 为正整数，给出下列 3 个算法关于问题规模 n 的时间复杂度。

(1) 算法 1：

```
void fun1(int n)
{   i = 1,k = 100;
    while (i <= n)
    {   k = k + 1;
        i += 2;
    }
}
```

(2) 算法 2：

```
void fun2(int b[],int n)
{   int i,j,k,x;
    for (i = 0;i < n - 1;i++)
    {   k = i;
        for (j = i + 1;j < n;j++)
            if (b[k]> b[j]) k = j;
        x = b[i];b[i] = b[k];b[k] = x;
    }
}
```

(3) 算法 3：

```
void fun3(int n)
{   int i = 0,s = 0;
    while (s <= n)
    {   i++;
        s = s + i;
    }
}
```

答：(1) 设 while 循环语句的执行次数为 $T(n)$，则有以下关系。

$i=2T(n)+1\leqslant n$，即 $T(n)\leqslant(n-1)/2=O(n)$。

(2) 算法中的基本运算语句是 if $(b[k]>b[j])$ $k=j$，其执行次数 $T(n)$ 为：

$$T(n)=\sum_{i=0}^{n-2}\sum_{j=i+1}^{n-1}1=\sum_{i=0}^{n-2}(n-i-1)=\frac{n(n-1)}{2}=O(n^2)$$

(3) 设 while 循环语句的执行次数为 $T(n)$，有：

$$s=1+2+\cdots+T(n)=\frac{T(n)(T(n)+1)}{2}\leqslant n$$

则 $T(n)=O(\sqrt{n})$。

12. 有以下递归算法用于对数组 $a[i..j]$ 的元素进行归并排序：

```
void mergesort(int a[],int i,int j)
{   int m;
    if (i!= j)
    {   m = (i + j)/2;
        mergesort(a,i,m);
        mergesort(a,m + 1,j);
        merge(a,i,j,m);
    }
}
```

求执行 mergesort$(a,0,n-1)$的时间复杂度。其中，merge(a,i,j,m)用于两个有序子序列 $a[i..m]$ 和 $a[m+1..j]$ 的合并，是非递归函数，它的时间复杂度为 O(合并的元素个数)。

答：设 mergesort$(a,0,n-1)$的执行时间为 $T(n)$，分析得到以下递归关系。

$$T(n)=\begin{cases}O(1) & n=1\\ 2T(n/2)+O(n) & n>1\end{cases}$$

其中，$O(n)$为 merge()所需的时间，设为 cn(c 为常量)。因此：

$$\begin{aligned}T(n)&=2T\left(\frac{n}{2}\right)+cn=2\left(2T\left(\frac{n}{2^2}\right)+\frac{cn}{2}\right)+cn=2^2T\left(\frac{n}{2^2}\right)+2cn=2^3T\left(\frac{n}{2^3}\right)+3cn\\&\quad\vdots\\&=2^kT\left(\frac{n}{2^k}\right)+kcn=2^kO(1)+kcn\end{aligned}$$

由于$\frac{n}{2^k}$趋近于 1，$k=\log_2 n$。所以 $T(n)=2^{\log_2 n}O(1)+cn\log_2 n=n+cn\log_2 n=O(n\log_2 n)$。

13. 描述一个集合的抽象数据类型 ASet，其中所有元素为正整数，集合的基本运算包括：

(1) 由整数数组 $a[0..n-1]$创建一个集合。

(2) 输出一个集合的所有元素。

(3) 判断一个元素是否在一个集合中。

(4) 求两个集合的并集。

(5) 求两个集合的差集。

（6）求两个集合的交集。

在此基础上设计集合的顺序存储结构，并实现各基本运算的算法。

答：抽象数据类型 ASet 的描述如下。

```
ADT ASet
{    数据对象: D = {dᵢ|0≤i≤n,n 为一个正整数}
     数据关系: 无.
     基本运算:
          createset(&s,a,n): 创建一个集合 s;
          dispset(s): 输出集合 s;
          inset(s,e): 判断 e 是否在集合 s 中;
          void add(s1,s2,s3): s3 = s1∪s2;               //求集合的并集
          void sub(s1,s2,s3): s3 = s1 - s2;              //求集合的差集
          void intersection(s1,s2,s3): s3 = s1∩s2;      //求集合的交集
}
```

设计集合的顺序存储结构类型如下：

```
typedef struct                   //集合结构体类型
{    int data[MaxSize];          //存放集合中的元素,其中 MaxSize 为常量
     int length;                 //存放集合中的实际元素个数
} Set;                           //将集合结构体类型用一个新类型名 Set 表示
```

采用 Set 类型的变量存储一个集合。对应的基本运算算法设计如下：

```
void createset(Set &s,int a[],int n)          //创建一个集合
{    int i;
     for (i = 0;i < n;i++)
          s.data[i] = a[i];
     s.length = n;
}
void dispset(Set s)                           //输出一个集合
{    int i;
     for (i = 0;i < s.length;i++)
          printf(" %d ",s.data[i]);
     printf("\n");
}
bool inset(Set s,int e)                       //判断 e 是否在集合 s 中
{    int i;
     for (i = 0;i < s.length;i++)
          if (s.data[i] == e)
               return true;
     return false;
}
void add(Set s1,Set s2,Set &s3)               //求集合的并集
{    int i;
     for (i = 0;i < s1.length;i++)            //将集合 s1 中的所有元素复制到 s3 中
          s3.data[i] = s1.data[i];
     s3.length = s1.length;
```

```
    for (i=0;i<s2.length;i++)          //将 s2 中不在 s1 中出现的元素复制到 s3 中
        if (!inset(s1,s2.data[i]))
        {   s3.data[s3.length]=s2.data[i];
            s3.length++;
        }
}
void sub(Set s1,Set s2,Set &s3)         //求集合的差集
{   int i;
    s3.length=0;
    for (i=0;i<s1.length;i++)          //将 s1 中不出现在 s2 中的元素复制到 s3 中
        if (!inset(s2,s1.data[i]))
        {   s3.data[s3.length]=s1.data[i];
            s3.length++;
        }
}
void intersection(Set s1,Set s2,Set &s3)    //求集合的交集
{   int i;
    s3.length=0;
    for (i=0;i<s1.length;i++)          //将 s1 中出现在 s2 中的元素复制到 s3 中
        if (inset(s2,s1.data[i]))
        {   s3.data[s3.length]=s1.data[i];
            s3.length++;
        }
}
```

1.3 补充练习题及参考答案



1.3.1 单项选择题

1. 数据结构是一门研究程序设计中数据的___①___以及它们之间的___②___和运算等的学科。

① A. 元素　　B. 计算方法　　C. 逻辑存储　　D. 映像

② A. 结构　　B. 关系　　C. 运算　　D. 算法

答：① A　② B。

2. 在数据结构中从逻辑上可以把数据结构分为________两类。

A. 动态结构和静态结构　　B. 紧凑结构和非紧凑结构

C. 线性结构和非线性结构　　D. 内部结构和外部结构

答：C。

3. 数据的逻辑结构是________关系的整体。

A. 数据元素之间逻辑　　B. 数据项之间逻辑

C. 数据类型之间　　D. 存储结构之间

答：A。

4. 下列说法中不正确的是________。

A. 数据元素是数据的基本单位

B. 数据项是数据中不可分割的最小可标识单位

C. 数据可由若干个数据元素构成

D. 数据项可由若干个数据元素构成

答：D。

5. 在计算机的存储器中表示数据时，物理地址和逻辑地址的相对位置相同并且是连续的，称之为________。

A. 逻辑结构　　B. 顺序存储结构

C. 链式存储结构　　D. 以上都对

答：B。

6. 在链式存储结构中，一个存储结点通常用于存储一个________。

A. 数据项　　B. 数据元素

C. 数据结构　　D. 数据类型

答：B。

7. 数据运算________。

A. 其执行效率与采用何种存储结构有关　　B. 是根据存储结构来定义的

C. 有算术运算和关系运算两大类　　D. 必须用程序设计语言来描述

答：A。

8. 数据结构在计算机内存中的表示是指________。

A. 数据的存储结构　　B. 数据结构

C. 数据的逻辑结构　　D. 数据元素之间的关系

答：A。

9. 在数据结构中，与所使用的计算机无关的是______。

A. 逻辑结构　　B. 存储结构

C. 逻辑结构和存储结构　　D. 物理结构

答：A。

10. 数据采用链式存储结构时要求________。

A. 每个结点占用一片连续的存储区域

B. 所有结点占用一片连续的存储区域

C. 结点的最后一个数据域是指针类型

D. 每个结点有多少个后继就设多少个指针域

答：A。

11. 以下叙述中正确的是________。

Ⅰ. 顺序存储方法仅适合存储线性结构的数据

Ⅱ. 算法分析的目的就是找出算法中输入和输出之间的关系

Ⅲ. 链式存储结构通过链指针表示数据元素之间的关系

Ⅳ. 抽象数据类型用于描述计算机求解问题的过程

A. 仅Ⅰ、Ⅲ　　B. 仅Ⅱ、Ⅳ　　C. 仅Ⅲ　　D. 仅Ⅳ

答：C。

12. 以下______不是算法的基本特性。

A. 可行性　　B. 长度有限

C. 在确定的时间内完成　　D. 确定性

答：选项 C 指的是有穷性，属算法的基本特性。本题的答案为 B。

13. 在计算机中算法指的是解决某一问题的有限运算序列，它必须具备输入、输出、________。

A. 可行性、可移植性和可扩充性　　B. 可行性、有穷性和确定性

C. 确定性、有穷性和稳定性　　D. 易读性、稳定性和确定性

答：B。

14. 下面关于算法的说法正确的是______。

A. 算法最终必须由计算机程序实现

B. 一个算法所花的时间等于该算法中每条语句的执行时间之和

C. 算法的可行性是指指令不能有二义性

D. 以上几个都是错误的

答：算法最终不一定由计算机程序实现，算法的确定性是指不能有二义性。本题的答案为 B。

15. 算法的时间复杂度与______有关。

A. 问题规模　　B. 计算机硬件性能

C. 编译程序质量　　D. 程序设计语言

答：A。

16. 算法分析的主要任务之一是分析______。

A. 算法是否具有较好的可读性

B. 算法中是否存在语法错误

C. 算法的功能是否符合设计要求

D. 算法的执行时间和问题规模之间的关系

答：D。

17. 算法分析的目的是________。

A. 找出数据结构的合理性　　B. 研究算法中的输入和输出关系

C. 分析算法的效率以求改进　　D. 分析算法的易读性和文档性

答：算法分析即算法效率分析，包括时间复杂度和空间复杂度分析，其目的是为了改进算法效率。本题的答案为 C。

18. 某算法的时间复杂度为 $O(n^2)$，表明该算法的________。

A. 问题规模是 n^2　　B. 执行时间等于 n^2

C. 执行时间与 n^2 成正比　　D. 问题规模与 n^2 成正比

答：C。

19. 某算法的时间复杂度为 $O(n)$，表示该算法的________。

A. 执行时间是 n　　B. 执行时间与 n 呈现线性增长关系

C. 执行时间不受 n 的影响　　D. 以上都不对

答：B。

20. 算法的空间复杂度是指________。

A. 算法中输入数据所占用的存储空间的大小

B. 算法本身所占用的存储空间的大小

C. 算法中占用的所有存储空间的大小

D. 算法中需要的临时变量所占用存储空间的大小

答：D。

1.3.2 填空题

1. 数据的逻辑结构是指________。

答：数据元素之间的逻辑关系。

2. 一个数据结构在计算机中的________称为存储结构。

答：映像。

3. 顺序存储方法是把逻辑上___①___存储在物理位置上___②___里；链式存储方法中结点间的逻辑关系是由___③___的。

答：①相邻的元素 ②相邻的存储单元 ③附加的指针域表示

4. 一个算法具有5个特性，即________、________、________、有输入和输出。

答：可行性、有穷性、确定性。

5. 在分析算法的时间复杂度时，通常认为算法的执行时间是________的函数。

答：问题规模。

6. 在算法描述中，当需要用一个形参直接改变对应的实参值时，该形参应该设计成________。

答：引用型参数。

7. 在算法描述时，在算法中对引用型参数的修改就是对相应________的修改。

答：实参。

8. 以下为各算法所有语句频度之和的表达式，其中时间复杂度相同的是________。

A. $T_A(n)=2n^3+3n^2+1000$　　B. $T_B(n)=n^3-n^2\log_2 n-1000$

C. $T_C(n)=n^2\log_2 n+n^2$　　D. $T_D(n)=n^2+1000$

答：$T_A(n)=O(n^3)$，$T_B(n)=O(n^3)$，$T_C(n)=O(n^2\log_2 n)$，$T_D(n)=O(n^2)$，所以时间复杂度相同的是A和B。

1.3.3 判断题

1. 判断以下叙述的正确性。

(1) 数据元素是数据的最小单位。

(2) 数据对象就是一组数据元素的集合。

(3) 任何数据结构都具备3个基本运算，即插入、删除和查找。

(4) 数据对象是由有限个类型相同的数据元素构成的。

(5) 数据的逻辑结构与各数据元素在计算机中如何存储有关。

(6) 如果数据元素值发生改变，则数据的逻辑结构也随之改变。

（7）逻辑结构相同的数据可以采用多种不同的存储方法。

（8）逻辑结构不相同的数据必须采用不同的存储方法来存储。

（9）数据的逻辑结构是指数据元素的各数据项之间的逻辑关系。

（10）抽象数据类型指的是某种特定的数据类型。

答：（1）错误。数据项是数据的最小单位。

（2）错误。这里未强调数据元素的性质相同。

（3）错误。如队列和栈等数据结构并不具备查找运算。

（4）正确。

（5）错误。

（6）错误。

（7）正确。

（8）错误。

（9）错误。数据的逻辑结构是指数据的各数据元素之间的逻辑关系。

（10）错误。

2．判断以下叙述的正确性。

（1）顺序存储方式只能用于存储线性结构。

（2）数据元素是数据的最小单位。

（3）算法可以用计算机语言描述，所以算法等同于程序。

（4）数据结构是带有结构的数据元素的集合。

（5）数据的逻辑结构是指数据元素之间的逻辑关系。

（6）数据逻辑结构、数据元素、数据项在计算机中的映像（或表示）分别称为存储结构、结点和数据域。

（7）数据的物理结构是指数据在计算机内的实际存储形式。

（8）算法 A 和算法 B 用于求解同一问题，算法 A 的最好时间复杂度为 $O(n)$，而算法 B 的最坏时间复杂度为 $O(n^3)$，则算法 A 好于算法 B。

（9）以下算法中没有循环语句，其时间复杂度为 $O(1)$：

```
int fun(int n)
{   if (n==1) return 1;
    else return n*fun(n-1);
}
```

（10）一个算法的空间复杂度为 $O(1)$，表示执行该算法不需要任何临时空间。

答：（1）错误。顺序存储方式也可用来存储树形结构，如完全二叉树可用一维数组存储。

（2）错误。数据元素是数据的基本单位，数据元素可以由数据项组成。

（3）错误。算法可以用计算机语言描述，但算法并不等同于程序，因为程序不一定满足有穷性，如 Windows 程序。

（4）正确。

（5）正确。

（6）正确。

（7）正确。

（8）错误。通常用两个算法的平均时间复杂度进行时间性能比较。

（9）错误。时间复杂度为 $O(n)$。

（10）错误。一个算法的空间复杂度为 $O(1)$，表示执行该算法所需要的临时空间大小与问题规模无关。

1.3.4 简答题

1. 什么是存储实现？什么是运算实现？

答：存储实现就是设计数据的存储结构，是建立数据的机内表示，包括两个部分，即数据元素和数据元素之间关系的存储。在某种存储实现的基础上各种运算的具体实现称为运算实现，运算实现的核心是设计实现某一运算的处理步骤，即算法设计。

2. 算法的时间复杂度反映的是算法的绝对执行时间吗？两个时间复杂度都为 $O(n^2)$ 的算法，对于相同的问题规模 n，它们的绝对执行时间一定相同吗？

答：算法的时间复杂度反映的是算法执行时间的数量级，并不是指绝对执行时间。两个时间复杂度都为 $O(n^2)$ 的算法，对于相同的问题规模 n，它们的绝对执行时间也不一定相同。

3. 设有算法如下：

```
int Find(int a[ ],int n,int x)
{   int i;
    for (i=0;i<n;i++)
        if (a[i]==x)
            return i;
    return -1;
}
```

成功找到 x 的最好和最坏时间复杂度是多少？

答：在最好情况下，$a[0]=x$，比较一次，所以最好时间复杂度为 $O(1)$。在最坏情况下，$a[n-1]=x$，比较 n 次，所以最坏时间复杂度为 $O(n)$。

4. 当为解决某一问题而选择数据的存储结构时应从哪些方面考虑？

答：通常从两个方面考虑，即执行算法所需的存储空间和执行时间。

5. 按增长率由小至大的顺序排列下列各函数：

2^{100}，$(2/3)^n$，$(3/2)^n$，n^n，$n!$，2^n，$\log_2 n$，$n^{\log_2 n}$，$n^{3/2}$，$\sqrt{n}$

答：按增长率由小至大的顺序可排列如下。

$(2/3)^n < 2^{100} < \log_2 n < \sqrt{n} < n^{3/2} < n^{\log_2 n} < (3/2)^n < 2^n < n! < n^n$

6. 计算以下算法中的语句频度（不计 return 语句的返回）。

```
double rsum(double a[], int n)
{   if (n<=0)
```

```
        return a[0];
    else
        return rsum(a,n-1) + a[n-1];
}
```

答：设算法中语句的频度为 $T(n)$，当 $n\leqslant 0$ 时，直接返回，计 1 次，即 $T(n)=1$；否则，执行一次比较和一次相加运算，即 $T(n)=T(n-1)+2$。所以，$T(n)=T(n-1)+2=T(n-2)+2\times 2=\cdots=T(0)+2n=2n+1$。

1.3.5 算法设计及算法分析题

1. 分析以下算法的时间复杂度。

```
void fun(int n)
{   int y = 0;
    while (y * y <= n)
        y++;
}
```

答：设 while 语句的执行频度为 $T(n)$，每循环一次 y 增大 1，即 $y=T(n)$，有以下关系。

$$T(n)\times T(n)\leqslant n,\text{即 } T(n)^2\leqslant n,T(n)\leqslant\sqrt{n}=O(\sqrt{n})$$

该算法的时间复杂度为 $O(\sqrt{n})$。

2. 分析以下算法的时间复杂度。

```
void fun(int n)
{   int i,x = 0;
    for (i = 1;i<n;i++)
        for (j = i + 1;j<= n;j++)
            x++;
}
```

答：设基本运算 x++的执行次数为 $T(n)$，则有以下关系。

$$T(n)=\sum_{i=1}^{n-1}\sum_{j=i+1}^{n}1=\sum_{i=1}^{n-1}(n-i)=n(n-1)/2=O(n^2)$$

该算法的时间复杂度为 $O(n^2)$。

3. 分析以下算法的时间复杂度。

```
void fun(int n)
{   int s = 0,i,j,k;
    for (i = 0;i<= n;i++)
        for (j = 0;j<= i;j++)
            for (k = 0;k<j;k++)
                s++;
}
```

答：该算法的基本运算是语句 s++，其频度如下。

$$T(n)=\sum_{i=0}^{n}\sum_{j=0}^{i}\sum_{k=0}^{j-1}1=\sum_{i=0}^{n}\sum_{j=0}^{i}(j-1-0+1)=\sum_{i=0}^{n}\sum_{j=0}^{i}j$$
$$=\sum_{i=0}^{n}\frac{i(i+1)}{2}=\frac{1}{2}\left(\sum_{i=0}^{n}i^2+\sum_{i=0}^{n}i\right)$$
$$=\frac{1}{2}\left(\frac{n(n+1)(2n+1)}{6}+\frac{n(n+1)}{2}\right)$$
$$=\frac{2n^3+6n^2+4n}{12}=O(n^3)$$

该算法的时间复杂度为 $O(n^3)$。

4. 设 n 是偶数，试计算执行以下算法后 m 的值，并给出其时间复杂度。

```
void fun(int n)
{   int m = 0,i,j;
    for (i = 1;i <= n;i++)
        for (j = 2 * i;j <= n;j++)
            m++;
}
```

答：算法的基本运算为 m++，由于内循环为 $2*i\sim n$，即 i 的最大值满足 $2i\leqslant n$，$i\leqslant n/2$，所以基本运算的频度如下。

$$T(n)=\sum_{i=1}^{n/2}\sum_{j=2i}^{n}1=\sum_{i=1}^{n/2}(n-2i+1)=n\times\frac{n}{2}-2\sum_{i=1}^{n/2}i+\frac{n}{2}=\frac{n^2}{4}$$

而 m 是从 0 开始的，所以算法执行后 $m=n^2/4$。该算法的时间复杂度为 $O(n^2)$。

5. 设 n 为正整数，分析以下算法中各语句的频度。

```
void fun(int n)
{   int i,j,k;
    for (i = 0;i < n;i++)                              //语句①
        for (j = 0;j < n;j++)                          //语句②
        {   c[i][j] = 0;                               //语句③
            for (k = 0;k < n;k++)                      //语句④
                c[i][j] = c[i][j] + a[i][k] * b[k][j]; //语句⑤
        }
}
```

解：语句①的频度为 $n+1$（i 从 0 到 $n-1$，共计 n 次，当 $i=n$ 时还要执行一次 $i<n$ 的比较，计一次，故总共 $n+1$ 次）。

语句②的频度为 $\sum_{i=0}^{n-1}(n+1)=n(n+1)$。

语句③的频度为 $\sum_{i=0}^{n-1}\sum_{j=0}^{n-1}1=n^2$。

语句④的频度为 $\sum_{i=0}^{n-1}\sum_{j=0}^{n-1}(n+1)=n^3+n^2$。

语句⑤的频度为 $\sum_{i=0}^{n-1}\sum_{j=0}^{n-1}\sum_{k=0}^{n-1}1=n^3$。

6. 设 n 为 3 的倍数，分析以下算法的时间复杂度。

```
void fun(int n)
{    int i,j,x,y;
     for (i = 1;i <= n;i++)
          if (3 * i <= n)
               for (j = 3 * i;j <= n;j++)
               {    x++;
                    y = 3 * x + 2;
               }
}
```

解：该算法中的基本运算是 x++和 $y=3*x+2$ 语句。对于最外层的 for 循环，其执行频度为 $n+1$，但对于里层的 for 循环，只在 $3i\leqslant n$（即 $i\leqslant n/3$）时才执行，故基本运算的执行频度$=\sum_{i=1}^{n//3}\sum_{j=3i}^{n}1=\sum_{i=1}^{n/3}(n-3i+1)=\frac{n(n-1)}{6}=O(n^2)$。本算法的时间复杂度为 $O(n^2)$。

7. 设计一个尽可能高效的算法，在长度为 n 的一维实型数组 $a[0..n-1]$中查找值最大的元素 max 和值最小的元素 min，并分析算法的最好、最坏和平均情况下元素的比较次数。

答：对应的算法如下。

```
void MaxMin(double a[],int n,int &max,int &min)
{    int i;
     max = min = a[0];
     for (i = 1;i < n;i++)
          if (a[i]> max)
               max = a[i];
          else if (a[i]< min)
               min = a[i];
}
```

该算法的最好、最坏和平均情况下元素的比较次数分别是 $n-1$、$2(n-1)$和 $3(n-1)/2$。

该算法的时间主要花费在元素的比较上。最好情况是 a 中的元素递增排列，元素比较次数为 $n-1$。最坏情况是 a 中的元素递减排列，元素比较次数为 $2(n-1)$。

对于平均情况，a 中有一半的元素比 max 大，$a[i]>$max 比较执行 $n-1$ 次，$a[i]<$min 比较执行$(n-1)/2$ 次，因此平均元素比较次数为 $3(n-1)/2$。

8. 设计尽可能高效的算法求一个整数数组中的最大元素和次大元素。并分析该算法的最好和最坏情况下的元素比较次数和时间复杂度。

解：对应的算法如下。

```
void maxmin(int a[],int n,int &max1,int &max2)
{    int i;
     max1 = (a[0]> a[1])?a[0]:a[1]; //求 a[0]、a[1]中的较大者
     max2 = (a[0]> a[1])?a[1]:a[0]; //求 a[0]、a[1]中的较小者
     for (i = 2;i < n;i++)
          if (a[i]> max1)
```

```
        {   max2 = max1;
            max1 = a[i];
        }
        else if (a[i]> max2)
            max2 = a[i];
}
```

算法中 max1 保存最大元素，max2 保存次大元素。首先通过两次比较求出 a[0]、a[1] 中的最大元素和次大元素，再 for 循环扫描其余元素。

在最好的情况下，for 循环中的 if 条件($a[i]$>max1)总是满足的，即数组 a 中的元素递增排列，这样不会执行 else 语句部分，for 循环中总的元素比较次数为 $n-2$ 次。这样总共的元素比较次数$=n-2+2=n$ 次。

在最坏的情况下，for 循环中的 if 条件($a[i]$>max1)总是不满足的，即数组 a 中的元素递减排列，这样就会执行 else 语句部分，for 循环中总的元素比较次数为 $2(n-2)$次。这样总共的元素比较次数$=2n-2$ 次。

本算法的时间复杂度为 $O(n)$。

9. 分析以下算法的时间复杂度(其中问题规模 $n=j-i+1$)。

```
void fun(ElemType A[], int i, int j, ElemType &max, ElemType &min)
{   int mid;
    ElemType gmax, gmin, hmax, hmin;
    if (i == j)
    {   max = min = A[i];
        return;
    }
    if (i == j - 1)
    {   if (A[i]< A[j])
        {   max = A[j];min = A[i];}
        else
        {   max = A[i];min = A[j];}
        return;
    }
    mid = (i + j)/2;
    fun(A, i, mid, gmax, gmin);
    fun(A, mid + 1, j, hmax, hmin);
    max = (gmax > hmax?gmax:hmax);
    min = (gmin < hmin?gmin:hmin);
}
```

解：本算法采用递归方法求一维数组 $A[i..j]$中的最大元素 max 和最小元素 min。本算法的时间主要花在元素的比较上，设 $T(n)$表示本算法中的比较运算次数(设 $n=j-i+1$)，因此有以下递推式：

$T(1)=0$　　　　$n=1$(即满足 $i==j$ 条件，没有元素比较)

$T(2)=1$　　　　$n=2$(即满足 $i==j-1$ 条件，有一次元素比较)

$T(n)=T(\lfloor n/2 \rfloor)+T(\lceil n/2 \rceil)+2$　　　　$n>2$(两次递归调用，求 max 和 min 的两次元素比较)

当 n 是 2 的幂时（即存在正整数 k，使得 $n=2^k$）有：

$$\begin{aligned} T(n) &= 2\times T(n/2)+2 \\ &= 2\times(2\times T(n/2^2)+2)+2 = 2^2\times T(n/2^2)+2^2+2 \\ &\vdots \\ &= 2^{k-1}\times T(2)+\sum_{i=1}^{k-1}2^i = 2^{k-1}+2^k-2 = \frac{3}{2}n-2 = O(n) \end{aligned}$$

本算法的时间复杂度为 $O(n)$。

10. 设有算法如下：

```
int Find(ElemType a[ ], int s, int t, ElemType x)
{   int m = (s + t)/2;
    if (s <= t)
    {   if (a[m] == x)
            return m;
        else if (x < a[m])
            return Find(a, s, m - 1, x);
        else
            return Find(a, m + 1, t, x);
    }
    return - 1;
}
```

分析执行 Find$(a,0,n-1,x)$的时间复杂度。

答：设 Find$(a,0,n-1,x)$的执行时间为 $T(n)$，则有以下递推式。

$$T(n)=\begin{cases}1 & \text{当 } n=1 \\ T(n/2)+1 & \text{当 } n>1\end{cases}$$

不妨设 $n=2^k$，即 $k=\log_2 n$。

则

$$\begin{aligned} T(n) &= T(n/2)+1 = T(n/2^2)+2 = T(n/2^3)+3 \\ &\vdots \\ &= T(n/2^k)+k \\ &= 1+k = \log_2 n+1 = O(\log_2 n)。\end{aligned}$$

所以该算法的时间复杂度是 $O(\log_2 n)$。

第2章 线性表



2.1 本章知识体系

1. 知识结构图

本章的知识结构如图 2.1 所示。

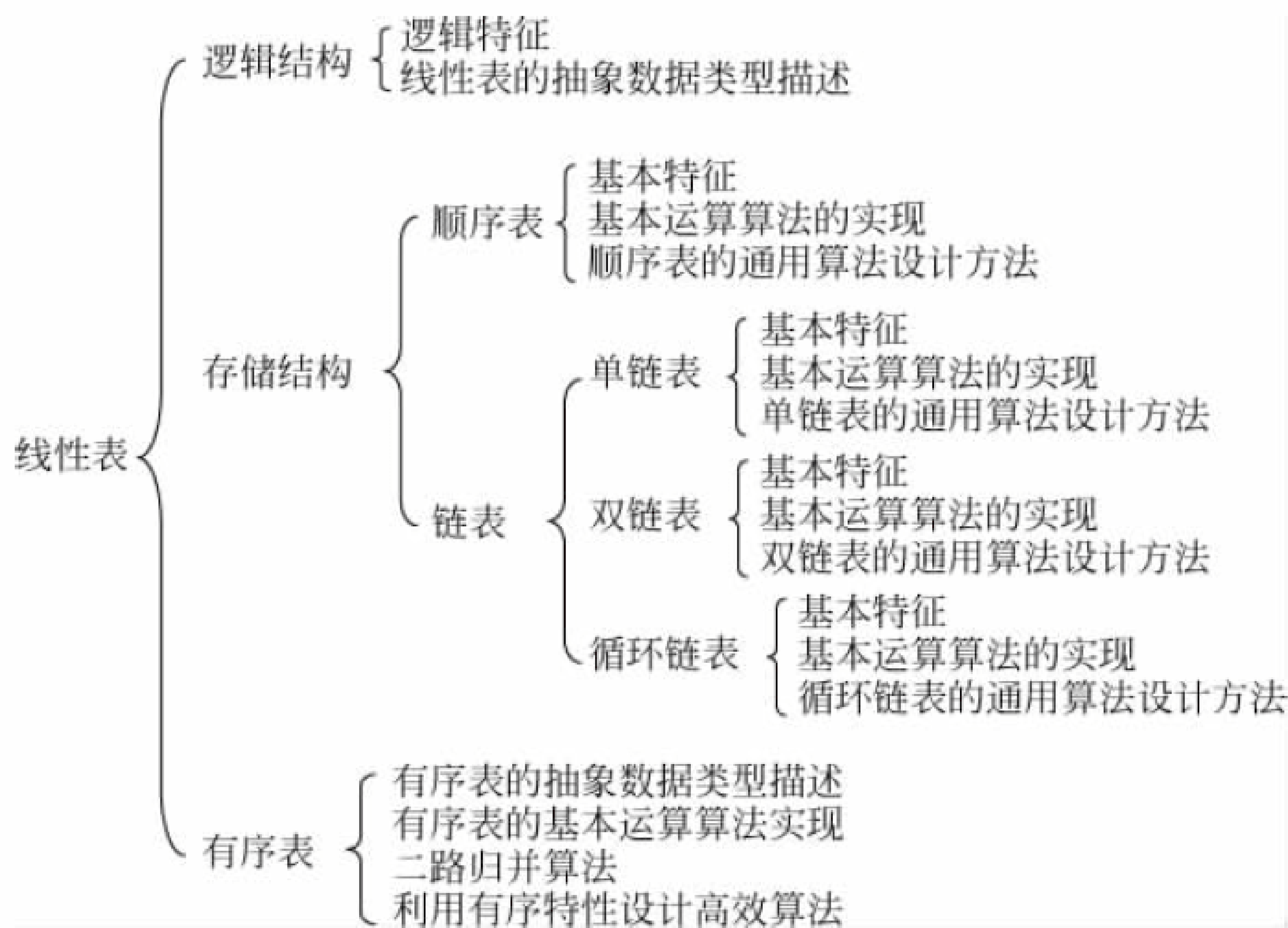


图 2.1　第 2 章知识结构图

2. 基本知识点

(1) 线性表的顺序存储结构和链式存储结构的优缺点。

(2) 顺序表的插入和删除操作过程及其实现。

(3) 单链表的查找、插入和删除操作过程及其实现。

(4) 双链表的查找、插入和删除操作过程及其实现。

(5) 循环链表的查找、插入和删除操作过程及其实现。

(6) 有序表的二路归并算法的思路及其实现算法，以及该算法的时间复杂度分析。

(7) 利用线性表求解复杂的应用问题。

3. 要点归纳

(1) 线性表是由 $n(n\geqslant 0)$ 个数据元素组成的有限序列，所有元素的性质相同，元素之间呈现线性关系，即除开始元素以外，每个元素只有唯一的前驱，除终端元素以外，每个元素只有唯一的后继。

(2) 在线性表中，通过序号来唯一标识一个元素，所以同一个线性表中可以存在值相同的元素。

(3) 顺序表采用数组存放元素，既可以顺序查找，也可以随机查找(对于给定的序号 i，在常量时间内找到对应的元素值)。

(4) 分配给顺序表的所有内存单元地址必须是连续的。

(5) 当从一个长度为 n 的顺序表中删除第 i 个元素($1\leqslant i\leqslant n$)时需向前移动 $n-i$ 个元素,所以删除算法的时间复杂度为 $O(n)$。

(6) 在一个长度为 n 的顺序表中插入第 i 个元素($1\leqslant i\leqslant n+1$)时需向后移动 $n-i+1$ 个元素,所以插入算法的时间复杂度为 $O(n)$。

(7) 链表由若干内存结点构成,结点的次序由地址确定,通过指针域反映数据的逻辑关系。

(8) 一个链表的所有结点的地址既可以连续,也可以不连续。

(9) 对链表只能顺序查找,不能随机查找,即给定序号 i,不能在常量时间内找到对应的结点。

(10) 对链表插入或删除结点不需要移动结点,只需要调整相应结点的指针域。

(11) 在单链表中存储每个结点有两个域,一个是数据域,另一个是指针域,指针域指向该结点的后继结点。

(12) 在带头结点的单链表中,通常用头结点指针标识整个单链表;在不带头结点的单链表中,通常用首结点指针标识整个单链表。

(13) 单链表只能按从前向后一个方向遍历。

(14) 在单链表中,插入一个新结点需要找到插入位置的前驱结点,通过修改两个指针域来实现。如插入一个新结点作为第 i 个结点,需要查找到第 $i-1$ 个结点 p,然后在结点 p 的后面插入新结点。

(15) 在单链表中,删除一个结点需要找到该结点的前驱结点,只需要修改一个指针域。如删除第 i 个结点,需要查找到第 $i-1$ 个结点 p,然后删除结点 p 的后继结点。

(16) 双链表可以按从前向后、从后向前两个方向遍历。

(17) 在双链表中,插入一个新结点需要找到插入位置的前驱结点或者后继结点,通过修改 4 个指针域来实现。如插入一个新结点作为第 i 个结点,需要查找到第 $i-1$ 个结点 p,然后在结点 p 的后面插入新结点;或者查找到第 i 个结点 q,然后在结点 q 的前面插入新结点。

(18) 在双链表中,删除一个结点通过该结点就可以直接实现,只需要修改两个指针域。如删除第 i 个结点,只需要查找到第 i 个结点 p,然后通过修改其前驱和后继结点的相应指针域来删除它。

(19) 循环链表分为循环单链表和循环双链表,循环单链表的结点构成一个查找环路,循环双链表的结点构成两个查找环路。

(20) 在循环单链表中没有指针域为空的结点。

(21) 在循环双链表中可以通过 $O(1)$ 的时间找到尾结点,删除它的时间复杂度为 $O(1)$。

(22) 线性表除了顺序表和链表两类存储结构以外,还可以设计成静态链表,静态链表采用静态空间分配方式,其中元素采用链表方式操作。静态链表不再具有随机查找特性。

(23) 有序表是一种按元素值有序排列的线性表,可以采用顺序表或链表存储。

(24) 长度分别为 n、m 的两个有序表采用二路归并方法合并成的一个有序表的时间复杂度为 $O(n+m)$,这是一种高效的方法。

2.2 教材中的练习题及参考答案

1. 简述线性表的两种存储结构的主要特点。

答：线性表的两种存储结构分别是顺序存储结构和链式存储结构。顺序存储结构的主要特点如下：

(1) 数据元素中只有自身的数据域，没有关联指针域，因此顺序存储结构的存储密度较大。

(2) 顺序存储结构需要分配一整块比较大的存储空间，所以存储空间的利用率较低。

(3) 逻辑上相邻的两个元素在物理上也是相邻的，通过元素的逻辑序号可以直接获取其元素值，即具有随机存取特性。

(4) 插入和删除操作会引起大量元素的移动。

链式存储结构的主要特点如下：

(1) 数据结点中除自身的数据域以外还有表示逻辑关系的指针域，因此链式存储结构比顺序存储结构的存储密度小。

(2) 链式存储结构的每个结点是单独分配的，每个结点的存储空间相对较小，所以存储空间利用率较高。

(3) 在逻辑上相邻的结点在物理上不一定相邻，因此不具有随机存取特性。

(4) 插入和删除操作方便、灵活，不必移动结点，只需修改结点中的指针域即可。

2. 简述单链表设置头结点的主要作用。

答：对单链表设置头结点的主要作用如下。

(1) 对于带头结点的单链表，在单链表的任何结点之前插入结点或删除结点，所要做的都是修改前一个结点的指针域，因为任何结点都有前驱结点(若单链表没有头结点，则首结点没有前驱结点，在其前插入结点和删除该结点时操作复杂一些)，所以算法设计方便。

(2) 对于带头结点的单链表，在表空时也存在一个头结点，因此空表与非空表的处理是一样的。

3. 假设某个含有 n 个元素的线性表有以下运算：

Ⅰ. 查找序号为 $i(1\leqslant i\leqslant n)$的元素；

Ⅱ. 查找第一个值为 x 的元素；

Ⅲ. 插入新元素作为第一个元素；

Ⅳ. 插入新元素作为最后一个元素；

Ⅴ. 插入第 $i(2\leqslant i\leqslant n)$个元素；

Ⅵ. 删除第一个元素；

Ⅶ. 删除最后一个元素；

Ⅷ. 删除第 $i(2\leqslant i\leqslant n)$个元素。

现设计该线性表的以下存储结构：

① 顺序表；

② 带头结点的单链表；

③ 带头结点的循环单链表；
④ 不带头结点仅有尾结点指针标识的循环单链表；
⑤ 带头结点的双链表；
⑥ 带头结点的循环双链表。
指出各种存储结构中对应运算算法的时间复杂度。
答：各种存储结构对应运算的时间复杂度如表 2.1 所示。

表 2.1　各种存储结构对应运算的时间复杂度

	Ⅰ	Ⅱ	Ⅲ	Ⅳ	Ⅴ	Ⅵ	Ⅶ	Ⅷ
①	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(n)$
②	$O(n)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$
③	$O(n)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$
④	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$
⑤	$O(n)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$
⑥	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$

4. 对于顺序表 L，指出以下算法的功能。

```
void fun(SqList *&L)
{   int i,j=0;
    for (i=1;i<L->length;i++)
        if (L->data[i]>L->data[j])
            j=i;
    for (i=j;i<L->length-1;i++)
        L->data[i]=L->data[i+1];
    L->length--;
}
```

答：该算法的功能是在顺序表 L 中查找第一个值最大的元素，并删除该元素。
5. 对于顺序表 L，指出以下算法的功能。

```
void fun(SqList *&L,ElemType x)
{   int i,j=0;
    for (i=1;i<L->length;i++)
        if (L->data[i]<=L->data[j])
            j=i;
    for (i=L->length;i>j;i--)
        L->data[i]=L->data[i-1];
    L->data[j]=x;
    L->length++;
}
```

答：在顺序表 L 中查找最后一个值最小的元素，在该位置上插入一个值为 x 的元素。

6. 有人设计以下算法用于删除整数顺序表 L 中所有值在$[x,y]$范围内的元素，该算法显然不是高效的，请设计一个同样功能的高效算法。

```
void fun(SqList *&L,ElemType x)
{   int i,j;
    for (i=0;i<L->length;i++)
        if (L->data[i]>=x && L->data[i]<=y)
        {   for (j=i;j<L->length-1;j++)
                L->data[j]=L->data[j+1];
            L->length--;
        }
}
```

答：该算法在每次查找到 x 元素时都通过移动来删除它，时间复杂度为 $O(n^2)$，显然它不是高效的算法。实现同样功能的算法如下：

```
void fun(SqList *&L,ElemType x,ElemType y)
{   int i,k=0;
    for (i=0;i<L->length;i++)
        if (!(L->data[i]>=x && L->data[i]<=y))
        {   L->data[k]=L->data[i];
            k++;
        }
    L->length=k;
}
```

该算法(思路参见《教程》例 2.3 的解法一)的时间复杂度为 $O(n)$，是一种高效的算法。

7. 设计一个算法，将元素 x 插入到一个有序(从小到大排序)顺序表的适当位置，并保持有序性。

解：通过比较在顺序表 L 中找到插入 x 的位置 i，将该位置及后面的元素均后移一个位置，将 x 插入位置 i 中，最后将 L 的长度增 1。对应的算法如下：

```
void Insert(SqList *&L,ElemType x)
{   int i=0,j;
    while (i<L->length && L->data[i]<x) i++;
    for (j=L->length-1;j>=i;j--)
        L->data[j+1]=L->data[j];
    L->data[i]=x;
    L->length++;
}
```

8. 假设一个顺序表 L 中的所有元素为整数，设计一个算法调整该顺序表，使其中所有小于零的元素放在所有大于等于零的元素的前面。

解：先让 i、j 分别指向顺序表 L 的第一个元素和最后一个元素。当 $i<j$ 时循环，i 从前向后扫描顺序表 L，找大于等于 0 的元素，j 从后向前扫描顺序表 L，找小于 0 的元素，当 $i<j$ 时将两元素交换(思路参见《教程》例 2.4 的解法一)。对应的算法如下：

```
void fun(SqList *&L)
{   int i=0,j=L->length-1;
    while (i<j)
    {   while (L->data[i]<0) i++;
        while (L->data[j]>=0) j--;
        if (i<j)          //L->data[i]与L->data[j]交换
            swap(L->data[i],L->data[j]);
    }
}
```

9. 对于不带头结点的单链表 $L1$，其结点类型为 LinkNode，指出以下算法的功能。

```
void fun1(LinkNode *&L1,LinkNode *&L2)
{   int n=0,i;
    LinkNode *p=L1;
    while (p!=NULL)
    {   n++;
        p=p->next;
    }
    p=L1;
    for (i=1;i<n/2;i++)
        p=p->next;
    L2=p->next;
    p->next=NULL;
}
```

答：对于含有 n 个结点的单链表 $L1$，将 $L1$ 拆分成两个不带头结点的单链表 $L1$ 和 $L2$，其中 $L1$ 含有原来的前 $n/2$ 个结点，$L2$ 含有余下的结点。

10. 在结点类型为 DLinkNode 的双链表中给出将 p 所指结点(非尾结点)与其后继结点交换的操作。

答：将 p 所指结点(非尾结点)与其后继结点交换的操作如下。

```
q=p->next;                    //q指向结点p的后继结点
if (q->next!=NULL)            //从链表中删除结点p
    q->next->prior=p;
p->next=q->next;
p->prior->next=q;             //将结点q插入到结点p的前面
q->prior=p->prior;
q->next=p;
p->prior=q;
```

11. 有一个线性表$(a_1,a_2,\cdots,a_n)$，其中 $n\geqslant 2$，采用带头结点的单链表存储，头指针为 L，每个结点存放线性表中的一个元素，结点类型为(data，next)，现查找某个元素值等于 x 的结点指针，若不存在这样的结点返回 NULL。分别写出下面 3 种情况的查找语句，要求时间尽量少。

(1) 线性表中的元素无序。

(2) 线性表中的元素按递增有序排列。

(3) 线性表中的元素按递减有序排列。

答:(1) 元素无序时的查找语句如下。

```
p = L -> next;
while (p!= NULL && p -> data!= x)
    p = p -> next;
if (p == NULL) return NULL;
else return p
```

(2) 元素按递增有序排列时的查找语句如下。

```
p = L -> next;
while (p!= NULL && p -> data < x)
    p = p -> next;
if (p == NULL || p -> data > x) return NULL;
else return p;
```

(3) 元素按递减有序排列时的查找语句如下。

```
p = L -> next;
while (p!= NULL && p -> data > x)
    p = p -> next;
if (p == NULL || p -> data < x) return NULL;
else return p;
```

12. 设计一个算法,将一个带头结点的数据域依次为 a_1、a_2、…、a_n($n \geqslant 3$)的单链表的所有结点逆置,即第 1 个结点的数据域变为 a_n,第 2 个结点的数据域变为 a_{n-1},…,尾结点的数据域变为 a_1。

解:首先让 p 指针指向首结点,将头结点的 next 域设置为空,表示新建的单链表为空表。用 p 扫描单链表的所有数据结点,将结点 p 采用头插法插入到新建的单链表中。对应的算法如下:

```
void Reverse(LinkNode *&L)
{   LinkNode *p = L -> next, *q;
    L -> next = NULL;
    while (p!= NULL)            //扫描所有的结点
    {   q = p -> next;          //q 临时保存 p 结点的后继结点
        p -> next = L -> next;  //总是将 p 结点作为首结点插入
        L -> next = p;
        p = q;                  //让 p 指向下一个结点
    }
}
```

13. 一个线性表(a_1,a_2,…,a_n)($n>3$)采用带头结点的单链表 L 存储,设计一个高效算法求中间位置的元素(即序号为$\lfloor n/2 \rfloor$的元素)。

解:让 p、q 首先指向首结点,然后在 p 结点后面存在两个结点时循环,p 后移两个结

点，q 后移一个结点。当循环结束后，q 指向的就是中间位置的结点，对应的算法如下：

```
ElemType Midnode(LinkNode * L)
{   LinkNode * p = L -> next, * q = p;
    while (p -> next!= NULL && p -> next -> next!= NULL)
    {   p = p -> next -> next;
        q = q -> next;
    }
    return q -> data;
}
```

14. 设计一个算法在带头结点的非空单链表 L 中第一个最大值结点（最大值结点可能有多个）之前插入一个值为 x 的结点。

解：先在单链表 L 中查找第一个最大值结点的前驱结点 maxpre，然后在其后面插入值为 x 的结点。对应的算法如下：

```
void Insertbeforex(LinkNode * &L, ElemType x)
{   LinkNode * p = L -> next, * pre = L;
    LinkNode * maxp = p, * maxpre = L, * s;
    while (p!= NULL)
    {   if (maxp -> data < p -> data)
        {   maxp = p;
            maxpre = pre;
        }
        pre = p; p = p -> next;
    }
    s = (LinkNode * )malloc(sizeof(LinkNode));
    s -> data = x;
    s -> next = maxpre -> next;
    maxpre -> next = s;
}
```

15. 设有一个带头结点的单链表 L，结点的结构为(data，next)，其中 data 为整数元素，next 为后继结点的指针。设计一个算法，首先按递减次序输出该单链表中各结点的数据元素，然后释放所有结点占用的存储空间，并要求算法的空间复杂度为 $O(1)$。

解：先对单链表 L 的所有结点递减排序（思路参见《教程》例 2.8），再输出所有结点值，最后释放所有结点的空间。对应的算法如下：

```
void Sort(LinkNode * &L)                //对单链表 L 递减排序
{   LinkNode * p, * q, * pre;
    p = L -> next -> next;              //p 指向第 2 个数据结点
    L -> next -> next = NULL;
    while (p!= NULL)
    {   q = p -> next;
        pre = L;
        while (pre -> next!= NULL && pre -> next -> data > p -> data)
            pre = pre -> next;
```

```
            p->next=pre->next;              //在结点 pre 之后插入 p 结点
            pre->next=p;
            p=q;
        }
    }
    void fun(LinkNode *&L)                  //完成本题的算法
    {   printf("排序前单链表 L:");
        DispList(L);                        //调用基本运算算法
        Sort(L);
        printf("排序后单链表 L:");
        DispList(L);                        //调用基本运算算法
        printf("释放单链表 L\n");
        DestroyList(L);                     //调用基本运算算法
    }
```

16. 设有一个双链表 h,每个结点中除了有 prior、data 和 next 几个域以外,还有一个访问频度域 freq,在链表被启用之前,其值均初始化为零。每当进行 LocateNode(h,x)运算时,令元素值为 x 的结点中 freq 域的值加 1,并调整表中结点的次序,使其按访问频度的递减次序排列,以便使频繁访问的结点总是靠近表头。试写一个符合上述要求的 LocateNode 运算的算法。

解:在 DLinkNode 类型的定义中添加整型 freq 域,将该域初始化为 0。在每次查找到一个结点 p 时将其 freq 域增 1,再与它前面的一个结点 pre 进行比较,若 p 结点的 freq 域值较大,则两者交换,如此找一个合适的位置。对应的算法如下:

```
bool LocateNode(DLinkNode *h,ElemType x)
{   DLinkNode *p=h->next, *pre;
    while (p!=NULL && p->data!=x)
        p=p->next;                          //找 data 域值为 x 的结点 p
    if (p==NULL)                            //未找到的情况
        return false;
    else                                    //找到的情况
    {   p->freq++;                          //频度增 1
        pre=p->prior;                       //结点 pre 为结点 p 的前驱结点
        while (pre!=h && pre->freq<p->freq)
        {   p->prior=pre->prior;
            p->prior->next=p;               //交换结点 p 和结点 pre 的位置
            pre->next=p->next;
            if (pre->next!=NULL)            //若 p 结点不是尾结点
                pre->next->prior=pre;
            p->next=pre;pre->prior=p;
            pre=p->prior;                   //q 指向结点 p 的前驱结点
        }
        return true;
    }
}
```

17. 设 ha=($a_1,a_2,\cdots,a_n$)和 hb=($b_1,b_2,\cdots,b_m$)是两个带头结点的循环单链表，设计一个算法将这两个表合并为带头结点的循环单链表 hc。

解：先找到 ha 的尾结点 p，将结点 p 的 next 指向 hb 的首结点，再找到 hb 的尾结点 p，将其构成循环单链表。对应的算法如下：

```
void Merge(LinkNode *ha, LinkNode *hb, LinkNode *&hc)
{   LinkNode *p = ha->next;
    hc = ha;
    while (p->next!= ha)                //找到 ha 的尾结点 p
        p = p->next;
    p->next = hb->next;                 //将结点 p 的 next 指向 hb 的首结点
    while (p->next!= hb)
        p = p->next;                    //找到 hb 的尾结点 p
    p->next = hc;                       //构成循环单链表
    free(hb);                           //释放 hb 单链表的头结点
}
```

18. 设两个非空线性表分别用带头结点的循环双链表 ha 和 hb 表示，设计一个算法 Insert(ha,hb,i)，其功能是当 $i=0$ 时将 hb 插入到 ha 的前面；当 $i>0$ 时将 hb 插入到 ha 中第 i 个结点的后面；当 i 大于等于 ha 的长度时将 hb 插入到 ha 的后面。

解：利用带头结点的循环双链表的特点设计的算法如下。

```
void Insert(DLinkNode *&ha, DLinkNode *&hb, int i)
{   DLinkNode *p = ha->next, *post;
    int lena = 1, j;
    while (p->next!= ha)            //求出 ha 的长度 lena
    {   lena++;
        p = p->next;
    }
    if (i == 0)                     //将 hb 插入到 ha 的前面
    {   p = hb->prior;              //p 指向 hb 的尾结点
        p->next = ha->next;         //将结点 p 链到 ha 的首结点前面
        ha->next->prior = p;
        ha->next = hb->next;
        hb->next->prior = ha;       //将 ha 头结点与 hb 的首结点链起来
    }
    else if (i < lena)              //将 hb 插入到 ha 中间
    {   j = 1;
        p = ha->next;
        while (j < i)               //在 ha 中查找第 i 个结点 p
        {   p = p->next;
            j++;
        }
        post = p->next;             //post 指向 p 结点的后继结点
        p->next = hb->next;         //将 hb 的首结点作为 p 结点的后继结点
        hb->next->prior = p;
        hb->prior->next = post;     //将 post 结点作为 hb 尾结点的后继结点
        post->prior = hb->prior;
```

```
        }
        else                                        //将 hb 链到 ha 之后
        {   ha->prior->next=hb->next;              //ha->prior 指向 ha 的尾结点
            hb->next->prior=ha->prior;
            hb->prior->next=ha;
            ha->prior=hb->prior;
        }
        free(hb);                                   //释放 hb 头结点
    }
```

19. 用带头结点的单链表表示整数集合，完成以下算法并分析时间复杂度：

(1) 设计一个算法求两个集合 A 和 B 的并集运算，即 $C=A\cup B$，要求不破坏原有的单链表 A 和 B。

(2) 假设集合中的元素按递增排列，设计一个高效算法求两个集合 A 和 B 的并集运算，即 $C=A\cup B$，要求不破坏原有的单链表 A 和 B。

解：(1) 集合 A、B、C 分别用单链表 ha、hb、hc 存储。采用尾插法创建单链表 hc，先将 ha 单链表中的所有结点复制到 hc 中，然后扫描单链表 hb，将其中所有不属于 ha 的结点复制到 hc 中。对应的算法如下：

```
void Union1(LinkNode *ha,LinkNode *hb,LinkNode *&hc)
{   LinkNode *pa=ha->next,*pb=hb->next,*pc,*rc;
    hc=(LinkNode *)malloc(sizeof(LinkNode));
    rc=hc;
    while (pa!=NULL)                    //将 A 复制到 C 中
    {   pc=(LinkNode *)malloc(sizeof(LinkNode));
        pc->data=pa->data;
        rc->next=pc;
        rc=pc;
        pa=pa->next;
    }
    while (pb!=NULL)                    //将 B 中不属于 A 的元素复制到 C 中
    {   pa=ha->next;
        while (pa!=NULL && pa->data!=pb->data)
            pa=pa->next;
        if (pa==NULL)                   //pb->data 不在 A 中
        {   pc=(LinkNode *)malloc(sizeof(LinkNode));
            pc->data=pb->data;
            rc->next=pc;
            rc=pc;
        }
        pb=pb->next;
    }
    rc->next=NULL;                      //尾结点 next 域置为空
}
```

本算法的时间复杂度为 $O(m\times n)$，其中 m、n 为单链表 ha 和 hb 中的数据结点个数。

(2) 同样采用尾插法创建单链表 hc，并利用单链表的有序性，采用二路归并方法来提高

算法效率。对应的算法如下：

```
void Union2(LinkNode * ha,LinkNode * hb,LinkNode * &hc)
{   LinkNode * pa = ha -> next, * pb = hb -> next, * pc, * rc;
    hc = (LinkNode * )malloc(sizeof(LinkNode));
    rc = hc;
    while (pa!= NULL && pb!= NULL)
    {   if (pa -> data < pb -> data)            //将较小的结点 pa 复制到 hc 中
        {   pc = (LinkNode * )malloc(sizeof(LinkNode));
            pc -> data = pa -> data;
            rc -> next = pc;
            rc = pc;
            pa = pa -> next;
        }
        else if (pa -> data > pb -> data)       //将较小的结点 pb 复制到 hc 中
        {   pc = (LinkNode * )malloc(sizeof(LinkNode));
            pc -> data = pb -> data;
            rc -> next = pc;
            rc = pc;
            pb = pb -> next;
        }
        else                                    //相等的结点只复制一个到 hc 中
        {   pc = (LinkNode * )malloc(sizeof(LinkNode));
            pc -> data = pa -> data;
            rc -> next = pc;
            rc = pc;
            pa = pa -> next;
            pb = pb -> next;
        }
    }
    if (pb!= NULL) pa = pb;                     //让 pa 指向没有扫描完的单链表结点
    while (pa!= NULL)
    {   pc = (LinkNode * )malloc(sizeof(LinkNode));
        pc -> data = pa -> data;
        rc -> next = pc;
        rc = pc;
        pa = pa -> next;
    }
    rc -> next = NULL;                          //尾结点 next 域置为空
}
```

本算法的时间复杂度为 $O(m+n)$，其中 m、n 为单链表 ha 和 hb 中的数据结点个数。

20. 用带头结点的单链表表示整数集合，完成以下算法并分析时间复杂度：

(1) 设计一个算法求两个集合 A 和 B 的差集运算，即 $C=A-B$，要求算法的空间复杂度为 $O(1)$，并释放单链表 A 和 B 中不需要的结点。

(2) 假设集合中的元素按递增排列，设计一个高效算法求两个集合 A 和 B 的差集运算，即 $C=A-B$，要求算法的空间复杂度为 $O(1)$，并释放单链表 A 和 B 中不需要的结点。

解：集合 A、B、C 分别用单链表 ha、hb、hc 存储。由于要求空间复杂度为 $O(1)$，不能采

用复制方法,只能利用原来单链表中的结点重组产生结果单链表。

(1) 将 ha 单链表中所有在 hb 中出现的结点删除,然后将 hb 中的所有结点删除。对应的算法如下:

```
void Sub1(LinkNode *ha,LinkNode *hb,LinkNode *&hc)
{   LinkNode *prea=ha, *pa=ha->next, *pb, *p, *post;
    hc=ha;                          //将 ha 的头结点作为 hc 的头结点
    while (pa!=NULL)                //删除 A 中属于 B 的结点
    {   pb=hb->next;
        while (pb!=NULL && pb->data!=pa->data)
            pb=pb->next;
        if (pb!=NULL)               //pa->data 在 B 中,从 A 中删除结点 pa
        {   prea->next=pa->next;
            free(pa);
            pa=prea->next;
        }
        else
        {   prea=pa;                //prea 和 pa 同步后移
            pa=pa->next;
        }
    }
    p=hb; post=hb->next;            //释放 B 中的所有结点
    while (post!=NULL)
    {   free(p);
        p=post;
        post=post->next;
    }
    free(p);
}
```

本算法的时间复杂度为 $O(m\times n)$,其中 m、n 为单链表 ha 和 hb 中的数据结点个数。

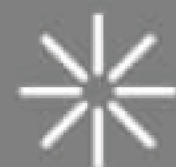
(2) 同样采用尾插法创建单链表 hc,并利用单链表的有序性,采用二路归并方法来提高算法效率,一边比较一边将不需要的结点删除。对应的算法如下:

```
void Sub2(LinkNode *ha,LinkNode *hb,LinkNode *&hc)
{   LinkNode *prea=ha, *pa=ha->next;      //pa 扫描 ha,prea 是 pa 结点的前驱结点指针
    LinkNode *preb=hb, *pb=hb->next;      //pb 扫描 hb,preb 是 pb 结点的前驱结点指针
    LinkNode *rc;                         //hc 的尾结点指针
    hc=ha;                                //ha 的头结点作为 hc 的头结点
    rc=hc;
    while (pa!=NULL && pb!=NULL)
    {   if (pa->data<pb->data)            //将较小的结点 pa 链到 hc 之后
        {   rc->next=pa;
            rc=pa;
            prea=pa;                      //prea 和 p 同步后移
            pa=pa->next;
        }
        else if (pa->data>pb->data)       //删除较大的结点 pb
```

```
            {   preb->next=pb->next;
                free(pb);
                pb=preb->next;
            }
            else                                //删除相等的 pa 结点和 pb 结点
            {   prea->next=pa->next;
                free(pa);
                pa=prea->next;
                preb->next=pb->next;
                free(pb);
                pb=preb->next;
            }
        }
        while (pb!=NULL)                        //删除 pb 余下的结点
        {   preb->next=pb->next;
            free(pb);
            pb=preb->next;
        }
        free(hb);                               //释放 hb 的头结点
        rc->next=NULL;                          //尾结点 next 域置为空
    }
```

本算法的时间复杂度为 $O(m+n)$，其中 m、n 为单链表 ha 和 hb 中的数据结点个数。

2.3 补充练习题及参考答案



2.3.1 单项选择题

1. 线性表是________。
   A. 一个有限序列，可以为空　　B. 一个有限序列，不可以为空
   C. 一个无限序列，可以为空　　D. 一个无限序列，不可以为空

答：A。

2. 在一个长度为 n 的顺序表中向第 i 个元素($1\leqslant i\leqslant n+1$)之前插入一个新元素时需要向后移动________个元素。
   A. $n-i$　　B. $n-i+1$　　C. $n-i-1$　　D. i

答：在第 i 个元素之前插入一个新元素时，$a_i \sim a_n$的元素都后移，这样后移的元素个数$=n-i+1$。所以本题选 B。

3. 一个顺序表所占用存储空间的大小与________无关。
   A. 顺序表的长度
   B. 顺序表中元素的数据类型
   C. 顺序表中元素各数据项的数据类型
   D. 顺序表中各元素的存放次序

答：D。

4. 顺序表具有随机存取特性指的是________。

A. 查找值为 x 的元素与顺序表中元素的个数 n 无关

B. 查找值为 x 的元素与顺序表中元素的个数 n 有关

C. 查找序号为 i 的元素与顺序表中元素的个数 n 无关

D. 查找序号为 i 的元素与顺序表中元素的个数 n 有关

答：C。

5. 顺序表和链表相比存储密度较大，这是因为________。

A. 顺序表的存储空间是预先分配的

B. 顺序表不需要增加指针来表示元素之间的逻辑关系

C. 链表的所有结点是连续的

D. 顺序表的存储空间是不连续的

答：B。

6. 链表不具有的特点是________。

A. 可随机访问任一元素　　B. 插入、删除不需要移动元素

C. 不必事先估计存储空间　　D. 所需空间与线性表长度成正比

答：链表不同于顺序表，不具有随机存取特性，所以本题选 A。

7. 当线性表采用链式存储结构时，各结点之间的地址________。

A. 必须是连续的　　B. 一定是不连续的

C. 部分地址必须是连续的　　D. 连续与否均可以

答：D。

8. 在线性表的下列存储结构中，读取指定序号的元素所花费时间最少的是________。

A. 单链表　　B. 双链表　　C. 循环链表　　D. 顺序表

答：顺序表具有随机存取特性，所以本题选 D。

9. 若线性表最常用的运算是存取第 i 个元素及其前驱元素值，则采用________存储方式节省时间。

A. 单链表　　B. 双链表　　C. 循环单链表　　D. 顺序表

答：顺序表适合于随机存取，所以本题选 D。

10. 对于含有 n 个元素的顺序表，其算法的时间复杂度为 $O(1)$ 的操作是________。

A. 将 n 个元素从小到大排序　　B. 删除第 i 个元素($1 \leqslant i \leqslant n$)

C. 查找第 i 个元素　　D. 在第 i 个元素之后插入一个元素

答：A 操作的时间复杂度为 $O(n^2)$ 或 $O(n\log_2 n)$；B 操作需要移动元素，时间复杂度为 $O(n)$；C 操作可以直接获得，时间复杂度为 $O(1)$；D 操作需要移动元素，时间复杂度为 $O(n)$。本题的答案为 C。

11. 设某个线性表有 n 个元素，在以下运算中，________在顺序表上实现比在链表上实现效率更高。

A. 输出第 i($1 \leqslant i \leqslant n$)个元素值　　B. 交换第 1 个元素与第 2 个元素的值

C. 顺序输出所有 n 个元素的值　　D. 求第 1 个值为 x 的元素的逻辑序号

答：A。

12. 以下关于单链表的叙述正确的是________。

Ⅰ. 结点除自身信息以外还包括指针域，存储密度小于顺序表

Ⅱ. 找第 i 个结点的时间为 $O(1)$

Ⅲ. 在插入、删除运算时不必移动结点

A. 仅Ⅰ、Ⅱ　　B. 仅Ⅱ、Ⅲ　　C. 仅Ⅰ、Ⅲ　　D. Ⅰ、Ⅱ、Ⅲ

答：C。

13. 通过含有 $n(n\geqslant1)$ 个元素的数组 a，采用头插法建立一个单链表 L，则 L 中结点值的次序________。

A. 与数组 a 的元素次序相同　　B. 与数组 a 的元素次序相反

C. 与数组 a 的元素次序无关　　D. 以上都不对

答：B。

14. 在单链表中，若 p 结点不是尾结点，在其后插入 s 结点的操作是________。

A. s—>next=p；p—>next=s；　　B. s—>next=p—>next；p—>next=s；

C. s—>next=p—>next；p=s；　　D. p—>next=s；s—>next=p；

答：B。

15. 在一个含有 n 个结点的有序单链表中插入一个新结点使得仍然有序，其算法的时间复杂度为________。

A. $O(\log_2 n)$　　B. $O(1)$　　C. $O(n^2)$　　D. $O(n)$

答：先要查找到插入结点的前驱结点，其时间复杂度为 $O(n)$。本题的答案为 D。

16. 在一个单链表中，删除 p 结点(非尾结点)之后的一个结点的操作是________。

A. p—>next=p　　B. p—>next—>next=p—>next

C. p—>next—>next=p　　D. p—>next=p—>next—>next

答：D。

17. 在单链表中删除 p 所指结点的后继结点，该算法的时间复杂度是________。

A. $O(1)$　　B. $O(\sqrt{n})$　　C. $O(\log_2 n)$　　D. $O(n)$

答：A。

18. 与单链表相比，双链表的优点之一是________。

A. 插入、删除操作更简单　　B. 可以进行随机访问

C. 可以省略表头指针或表尾指针　　D. 访问前后相邻结点更方便

答：D。

19. 在长度为 $n(n\geqslant1)$ 的双链表 L 中，在 p 所指结点之前插入一个新结点的时间复杂度为________。

A. $O(1)$　　B. $O(n)$　　C. $O(n^2)$　　D. $O(n\log_2 n)$

答：A。

20. 在长度为 $n(n\geqslant1)$ 的双链表 L 中，删除尾结点的时间复杂度为________。

A. $O(1)$　　B. $O(n)$　　C. $O(n^2)$　　D. $O(n\log_2 n)$

答：B。

21. 在长度为 $n(n\geqslant1)$ 的双链表 L 中，删除 p 所指结点的时间复杂度为________。

A. $O(1)$　　B. $O(n)$　　C. $O(n^2)$　　D. $O(n\log_2 n)$

答：A。

22. 在长度为 $n(n\geqslant 1)$ 的双链表 L 中，删除 p 所指结点的前驱结点的时间复杂度为________。

A. $O(1)$　　B. $O(n)$　　C. $O(n^2)$　　D. $O(n\log_2 n)$

答：A。

23. 在不带头结点的循环单链表 L 中，至少有一个结点的条件是＿①＿，尾结点 p 的条件是＿②＿。

A. L!=NULL　　B. L->next!=L

C. p==NULL　　D. p->next==L

答：①A ②D。

24. 在带头结点的循环单链表 L 中，至少有一个结点的条件是＿①＿，尾结点 p 的条件是＿②＿。

A. L->next!=NULL　　B. L->next!=L

C. p==NULL　　D. p->next==L

答：①B ②D。

25. 在只有尾结点指针 rear 没有头结点的非空循环单链表中，删除开始结点的时间复杂度为________。

A. $O(1)$　　B. $O(n)$　　C. $O(n^2)$　　D. $O(n\log_2 n)$

答：A。

26. 在长度为 $n(n\geqslant 1)$ 的循环双单链表 L 中，删除尾结点的时间复杂度为________。

A. $O(1)$　　B. $O(n)$　　C. $O(n^2)$　　D. $O(n\log_2 n)$

答：A。

27. 某线性表最常用的运算是在尾元素之后插入元素和删除尾元素，则以下________存储方式最节省运算时间。

A. 单链表　　B. 循环单链表　　C. 双链表　　D. 循环双链表

答：D。

28. 某线性表最常用的运算是在尾元素之后插入元素和删除开始元素，则以下________存储方式最节省运算时间。

A. 单链表　　B. 仅有头结点指针的循环单链表

C. 双链表　　D. 仅有尾结点指针的循环单链表

答：D。

29. 如果对含有 $n(n>1)$ 个元素的线性表的运算只有 4 种，即删除第一个元素、删除尾元素、在第一个元素前面插入新元素、在尾元素的后面插入新元素，则最好使用________。

A. 只有尾结点指针没有头结点的循环单链表

B. 只有尾结点指针没有头结点的非循环双链表

C. 只有首结点指针没有尾结点指针的循环双链表

D. 既有头指针也有尾指针的循环单链表

答：对于只有首结点指针没有尾结点指针的循环双链表，上述 4 种运算的时间复杂度均为 $O(1)$。本题的答案为 C。

30. 以下关于有序表的叙述正确的是________。

A. 有序表只能采用顺序表存储

B. 有序表中元素之间的关系是非线性关系

C. 有序表只能采用链表存储

D. 有序表既可以采用顺序表存储,也可以采用链表存储

答:D。

31. 将两个各有 n 个元素的递增有序顺序表归并成一个有序顺序表,其最少的元素比较次数是________。

A. n　　B. $2n-1$　　C. $2n$　　D. $n-1$

答:当一个有序顺序表的所有元素小于另一个有序顺序表时元素比较次数最少,为 n 次。本题的答案为 A。

32. 将两个长度分别为 n、m 的递增有序顺序表归并成一个有序顺序表,其元素最多的比较次数是________(MIN 表示取最小值)。

A. n　　B. $m+n$　　C. MIN(m,n)　　D. $m+n-1$

答:D。例如,两个表分别为(1,3,5,7)、(2,4,6),共需要 6 次比较。

2.3.2 填空题

1. 在线性表的顺序存储结构中,元素之间的逻辑关系是通过__①__决定的;在线性表的链式存储结构中,元素之间的逻辑关系是通过__②__决定的。

答:①物理存储位置　②指针域。

2. 在有 n 个元素的顺序表中删除任意一个元素所需移动元素的平均次数为________。

答:$(n-1)/2$。

3. 在一个长度为 $n(n\geqslant 1)$ 的顺序表中删除第 i 个元素 $(1\leqslant i\leqslant n)$ 时需向前移动________个元素。

答:需将 $a_{i+1}\sim a_n$ 元素均前移一位,共移动 $n-(i+1)+1=n-i$ 个元素。答案为 $n-i$。

4. 在有 n 个元素的顺序表中的任意位置插入一个元素所需移动元素的平均次数为________。

答:$n/2$。

5. 在长度为 n 的顺序表中,若删除第 $i(1\leqslant i\leqslant n)$ 个元素的概率是 p_i,则删除元素时平均移动元素的次数是________。

答:$\sum_{i=1}^{n} p_i(n-i)$ 。

6. 在长度为 n 的顺序表 L 中将所有值为 x 的元素替换成 y,该算法的时间复杂度为________。

答:$O(n)$。

7. 带头结点的单链表 L 为空的判定条件是________。

答:L->next==NULL。

8. 在一个单链表 L 中,已知 p 指向某个非尾结点,若要删除其后继结点,并释放其空间,则执行的操作是________。

答：$q=p->next$；$p->next=q->next$；free(q)。

9．对于一个具有 $n(n\geqslant 1)$ 个结点的单链表，插入一个尾结点的时间复杂度是________。

答：$O(n)$。需要查找尾结点的前驱结点，时间为 $O(n)$。

10．有一个含有 n 个元素的线性表，可以采用单链表或双链表存储，其主要的操作是插入和删除第一个元素，最好选择________存储结构。

答：单链表。就两种操作而言，两种链表均可，但单链表的存储密度较高。

11．以下算法是删除带头结点的单链表 L 中 p 所指的结点并释放它，请填空。

```
bool Delp(LinkNode *&L,LinkNode *p)
{   LinkNode *pre = L;
    while (pre->next!= p)
        ___①___;
    if (pre == NULL)
        return false;
    else
    {   ___②___;
        free(p);
        return true;
    }
}
```

答：①pre=pre->next　②pre->next=p->next。

12．求一个双链表长度的算法的时间复杂度为________。

答：$O(n)$。

13．某个含有 n 个元素的线性表可以采用单链表或双链表存储结构，但要求快速删除指定位置的结点，应采用________。

答：双链表。双链表中删除指定位置结点的时间为 $O(1)$。

14．对于双链表，删除 p 结点的前驱结点需要修改________个指针域。

答：2。

15．在一个双链表 L 中，若要在 p 结点(非尾结点)之前插入一个结点 s，则执行的操作是________。

答：在双链表中插入结点时要修改 4 个链域。本题的答案为 $s->prior=p->prior$；$p->prior->next=s$；$s->next=p$；$p->prior=s$；

16．在长度为 n 的循环单链表 L 中查找值最大的结点，其时间复杂度为________。

答：$O(n)$。

17．有一个长度为 n 的循环单链表 L，在 p 所指的结点之前插入一个新结点，其时间复杂度为________。

答：$O(n)$。需要查找结点 p 的前驱结点，时间为 $O(n)$。

18．在结点个数大于 1 的循环单链表中，指针 p 指向其中某个结点，当执行以下程序段后让指针 s 指向结点 p 的前驱结点，请填空。

```
s = p;
while (________) s = s->next;
```

答：s->next!=p。

19. 线性表采用某种链式存储结构，在该链表上删除尾结点的时间复杂度为 $O(1)$，则该链表是________。

答：循环双链表。

20. 在含有 n 个结点的循环双链表中要删除 p 所指的结点，其时间复杂度为________。

答：$O(1)$。

21. 两个长度分别为 m、n 的有序单链表，在采用二路归并算法产生一个有序单链表时，算法的时间复杂度为________。

答：$O(m+n)$。

22. 两个长度分别为 m、n 的有序顺序表，在采用二路归并算法产生一个有序顺序表时，最少的元素比较次数为________。

答：MIN(m,n)。例如，两个表分别为(1,2,3)、(4,5,6,7,8)，只需要比较 3 次。

2.3.3 判断题

1. 判断以下叙述的正确性。

(1) 在一个含有 $n(n\geqslant 1)$个元素的线性表中，所有元素值不能相同。

(2) 顺序表具有随机存取特性，所以查找值为 x 的元素的时间复杂度为 $O(1)$。

(3) 线性表(含 n 个元素)的基本运算之一是删除第 i 个元素，其中 i 的有效取值范围是 $0\leqslant i\leqslant n-1$。

(4) 顺序表采用一维数组存放线性表中的元素，所以顺序表与一维数组是等同的。

(5) 线性表的顺序存储表示属于静态结构，而链式存储表示属于动态结构。

(6) 在含有 n 个结点的单链表 L 中，将 p 所指结点(非首结点)与其前驱结点交换，时间复杂度为 $O(1)$。

(7) 在含有 n 个结点的双链表 L 中，将 p 所指结点(非首结点)与其前驱结点交换，时间复杂度为 $O(1)$。

(8) 在含有 n 个结点的双链表 L 中删除 p 所指的结点，时间复杂度为 $O(1)$。

(9) 在循环单链表中，从表中任一结点出发都可以通过指针前后移动操作遍历整个循环链表。

(10) 在含有 n 个结点的循环单链表 L 中删除 p 所指结点(非首结点)的前驱结点，时间复杂度为 $O(1)$。

答：(1) 错误。

(2) 错误。对应的时间复杂度为 $O(n)$。

(3) 错误。基本运算中的元素序号 i 是指逻辑序号，从 1 开始，这里 i 的有效取值范围是 $1\leqslant i\leqslant n$。

(4) 错误。顺序表要求所有元素连续存储，而一维数组不必这样，另外，一维数组只有存、取元素操作，而顺序表可以插入、删除元素，所以两者不能等同。

(5) 错误。所谓静态结构是采用固定大小的静态分配方式得到的结果，如“int a[10];”语句就是一种静态分配方式。所谓动态结构是采用动态分配方式得到的结果，如“int *p; p=(int *)malloc(n*sizeof(int));”语句就是一种动态分配方式。对于链式存储结构，通

常采用动态分配方式，而对于顺序存储结构，既可以采用静态分配方式，也可以采用动态分配方式，只不过为了简单，顺序存储结构通常采用静态分配方式。

(6) 错误。找结点 p 的前驱结点对应的时间为 $O(n)$。

(7) 正确。

(8) 正确。

(9) 错误。

(10) 错误。对应的时间为 $O(n)$。

2. 判断以下叙述的正确性。

(1) 分配给一个单链表所有结点的内存单元地址必须是连续的。

(2) 与顺序表相比，在链表中顺序访问所有结点，其算法的效率比较低。

(3) 从长度为 n 的顺序表中删除任何一个元素所需要的时间均为 $O(n)$。

(4) 向顺序表中插入一个元素平均要移动大约一半的元素。

(5) 空的单链表不含有任何结点。

(6) 如果单链表带有头结点，则任何插入操作都不会改变头结点指针的值。

(7) 在单链表中删除一个结点，首先需要找到该结点的前驱结点。

(8) 在双链表中删除一个结点(非尾结点)，需要修改 4 个指针域。

(9) 在循环单链表中没有为空的指针域。

(10) 要想在 $O(1)$ 的时间内访问尾结点，应采用循环单链表存储结构。

答：(1) 错误。分配给单链表的内存单元地址可以是不连续的。

(2) 错误。在顺序表和链表上顺序访问所有结点，时间复杂度均为 $O(n)$。

(3) 错误。在删除最后一个元素时，所需的时间都是 $O(1)$。但从长度为 n 的顺序表中删除任何一个元素，平均时间复杂度都是 $O(n)$。

(4) 正确。

(5) 错误。带头结点的单链表为空时仍有一个头结点。

(6) 正确。

(7) 正确。

(8) 错误。需要修改两个指针域。

(9) 正确。

(10) 错误。应采用循环双链表存储结构。

3. 判断以下叙述的正确性。

(1) 顺序存储结构的特点是存储密度大且插入、删除运算的效率高。

(2) 线性表的顺序存储结构总是优于链式存储结构。

(3) 由于顺序表需要一整块连续的存储空间，所以存储空间利用率高。

(4) 同一个线性表采用单链表和双链表存储时，单链表的存储密度高于双链表。

(5) 对于单链表来说，需要从头结点出发才能扫描表中的全部结点。

(6) 双链表的特点是很容易找任一结点的前驱和后继结点。

(7) 在双链表中删除一个结点 p 的前驱结点所花费的时间是 $O(n)$。

(8) 在双链表中删除第 i 个结点的前驱结点所花费的时间是 $O(n)$。

(9) 对于循环单链表来说，从表中任一结点出发都能扫描整个链表。

（10）利用非循环单链表实现的功能采用相应的循环单链表也可以实现。

答：（1）错误。顺序存储结构的优点是存储密度大，但插入、删除运算的效率低。

（2）错误。顺序和链式存储结构各有优缺点。

（3）错误。由于顺序表需要一整块连续的较大存储空间，当在存储器中出现比较多的碎片时，这些碎片空间可能得不到应用，所以存储空间利用率低。

（4）正确。

（5）正确。

（6）正确。

（7）错误。

（8）正确。需要查找第 i 个结点，对应的时间为 $O(n)$。

（9）正确。

（10）正确。

2.3.4 简答题

1. 线性表有两种存储结构，一是顺序表；二是链表，试问：

（1）如果有多个线性表同时共存，并且在处理过程中各表的长度会动态地发生变化，在此情况下应选用哪种存储结构？为什么？

（2）若线性表的元素个数基本稳定，且很少进行插入和删除，但要求快速存取线性表中指定序号的元素，那么应采用哪种存取结构？为什么？

答：（1）应选用链式存储结构。由于链式存储结构可以充分利用存储空间来存储线性表中的各数据元素，且其存储空间可以是连续的，也可以不连续；此外，在这种存储结构下，对元素进行插入和删除操作时都无须移动元素，而仅修改指针即可，所以很适用于线性表容量变化的情况。

（2）应选用顺序存储结构。由于顺序存储结构一旦确定了起始位置，线性表中的任何一个元素都可以进行随机存取，即存取速度较高。

2. 在线性表的以下链式存储结构中，若未知链表头结点的地址，仅已知 p 指针指向的结点，能否从中删除该结点？为什么？

（1）单链表。

（2）双链表。

（3）循环单链表。

答：（1）不能删除。因为无法找到 p 结点的前驱结点。

（2）能删除。由 p 结点的前驱指针找到其前驱结点，然后删除 p 结点。

（3）能删除。循环查找一周，可找到 p 结点的前驱结点，然后删除 p 结点。

3. 在单链表和双链表中能否从当前结点出发访问到任一结点？

答：在单链表中只能由当前结点访问其后继结点，但因其没有指向前驱的指针而无法访问其前驱结点。在双链表中，由于当前结点既有指向后继结点的指针，又有指向前驱结点的指针，所以在双向链表中可以由当前结点出发访问表中的任何一个结点。

4. 哪些链表从尾结点出发可以访问到链表中的任意结点？

答：循环单链表和循环双链表从尾指针出发可以访问到链表中的任意结点。

5. 为什么在不带头结点的循环单链表中设置尾指针比设置首指针更好?

答:尾指针是指向尾结点的指针,用它来标识循环单链表可以使得查找链表的首结点和尾结点都很方便。设循环单链表的尾指针为 rear,则首结点和尾结点的地址分别是 rear—> next 和 rear,查找时间都是 $O(1)$。

若用首指针来表示该链表,则查找尾结点的时间为 $O(n)$。

6. 带头结点的双链表和循环双链表相比有什么不同? 在何时使用循环双链表?

答:在带头结点的双链表中,尾结点的后继指针为 NULL,头结点的前驱指针不使用;在带头结点的循环双链表中,尾结点的后继指针指向头结点,头结点的前驱指针指向尾结点。当需要快速找到尾结点时可以使用循环双链表。

7. 以下算法用于统计带头结点的单链表 L 中结点值等于 x 的结点个数,其中存在错误,请指出错误的地方并修改为正确的算法。

```
int count(LinkNode * L,ElemType x)
{   int n = 0;
    while (L!= NULL)
    {   L = L -> next;
        if (L -> data == x) n++;
    }
    return n;
}
```

答:当 L 指向尾结点时 while 条件成立,再执行 $L=L->$next,则 $L=$NULL,此时 if 语句出现错误。修改后的算法如下:

```
int count(LinkNode * L,ElemType x)
{   LinkNode * p = L -> next;
    int n = 0;
    while (p!= NULL)
    {   if (p -> data == x) n++;
        p = p -> next;
    }
    return n;
}
```

8. 有以下关于单链表 L 的算法:

```
bool fun(LinkNode *&L,int i,ElemType e)
{   int j = 0;
    LinkNode * p = L, * s;
    while (j < i - 1 && p!= NULL)
    {   j++;
        p = p -> next;
    }
    if (p == NULL)
        return false;
    else
```

```
        {   s = (LinkNode * )malloc(sizeof(LinkNode));
            s -> data = e;
            s -> next = p -> next;
            p -> next = s;
            return true;
        }
    }
```

(1) 指出 fun(L,i,e)算法的功能。

(2) 当 $L=(1,2,3,4,5,6,7,8)$时，执行 fun($L,3,9$)后 L 的结果是什么？

答：(1) fun(L,i,e)算法的功能是在单链表 L 中插入元素值为 e 的第 i 个结点，成功插入时返回 true，否则返回 false。

(2) 当 $L=(1,2,3,4,5,6,7,8)$时，执行 fun($L,3,9$)后 $L=(1,2,9,3,4,5,6,7,8)$。

9. 有以下关于单链表 L 的算法：

```
void fun(LinkNode *&L)
{   LinkNode * prep = L, * p, * q;
    while (true)
    {   p = prep -> next;
        if (p == NULL) break;
        q = p -> next;
        if (q == NULL) break;
        p -> next = q -> next;
        q -> next = p;
        prep -> next = q;
        prep = p;
    }
}
```

(1) 指出 fun(L)算法的功能。

(2) 当 $L=(1,2,3,4,5,6,7,8,9)$时，执行 fun(L)后 L 的结果是什么？

答：(1) fun(L)算法的功能是从前向后将单链表 L 中的奇数序号结点和相邻的偶数序号结点交换。

(2) 当 $L=(1,2,3,4,5,6,7,8,9)$时，执行 fun($L,2,5$)后 $L=(2,1,4,3,6,5,8,7,9)$。

10. 有以下关于单链表 L 的算法：

```
void fun(LinkNode *&L,int i,int j)
{   int k = 0;
    LinkNode * pre, * p = L;
    while (k < i - 1 && p!= NULL)
    {   k++;
        p = p -> next;
    }
    if (p == NULL) return;
    pre = p;
    p = pre -> next;
```

```
    while (k<j && p!= NULL)
    {   pre->next = p->next;
        free(p);
        p = pre->next;
        k++;
    }
}
```

(1) 指出 fun(L,i,j)算法的功能。

(2) 当 L=(1,2,3,4,5,6,7,8)时,执行 fun(L,2,5)后 L 的结果是什么?

答:(1) fun(L,i,j)算法的功能是删除单链表 L 中从第 i 个结点到第 j 个结点的所有结点。

(2) 当 L=(1,2,3,4,5,6,7,8)时,执行 fun(L,2,5)后 L=(1,6,7,8)。

2.3.5 算法设计题

1.【顺序表算法】设计一个高效算法,将顺序表 L 中的所有元素逆置,要求算法的空间复杂度为 $O(1)$。

解:扫描顺序表 L 的前半部分元素,对于元素 L->data[i]($0 \leq i < L$->length/2),将其与后半部分对应的元素 L->data[L->length$-i-1$]进行交换。对应的算法如下:

```
void reverse(SqList *&L)
{   int i;
    for (i = 0;i<L->length/2;i++)                //交换一半的元素
        swap(L->data[i],L->data[L->length - i - 1]);
}
```

2.【顺序表算法】设计一个时间和空间两方面尽可能高效的算法,将顺序表 L 中存放的整数序列循环左移 p($0<p<n$,n 为 L 中的元素个数)个位置,即将 L 中的数据序列(X_0,X_1,…,X_{n_1})变换为(X_p,X_{p+1},…,X_{n-1},X_0,X_1,…,X_{p-1})。

解:设 R=(X_0,X_1,…,X_p,X_{p+1},…,X_{n-1}),其中 a=(X_0,X_1,…,X_{p-1})(共有 p 个元素),b=(X_p,…,X_{n-1})(共有 $n-p$ 个元素),并设 reverse(A)用于原地逆置数组 R,则 a 原地逆置后 a'变为(X_{p-1},…,X_1,X_0),b 原地逆置后 b'变为(X_{n-1},…,X_{p-1},X_p),也就是说 $a'b'$=(X_{p-1},…,X_1,X_0,X_{n-1},…,X_{p-1},X_p),再将 $a'b'$原地逆置变为(X_p,X_{p-1},…,X_{n-1},X_0,X_1,…,X_{p-1})即为所求,即 reverse(R)=reverse(reverse(a),reverse(b))。

对应的算法如下:

```
void reverse(SqList *&L,int m,int n)            //将 R[m..n]逆置
{   int i;
    for (i = 0;i<(n - m + 1)/2;i++)
        swap(L->data[m + i],L->data[n - i]);    //将 data[m + i]与 data[n - i]进行交换
}
bool creverse(SqList *&L,int p)                 //将 L 中的元素循环左移 p 个位置
```

```
{   if (p<=0 || p>=L->length)
        return false;
    else
    {   reverse(L,0,p-1);
        reverse(L,p,L->length-1);
        reverse(L,0,L->length-1);
        return true;
    }
}
```

其中，reverse(R,m,n)算法的时间复杂度为 $O(n-m)$，所以 creverse(R,n,p)算法的时间复杂度$=O(p)+O(n-p)+O(n)=O(n)$。另外，在 creverse(R,n,p)算法中只定义几个变量，所以空间复杂度为 $O(1)$。

3.【顺序表算法】若一个线性表采用顺序表 L 存储，其中所有元素为整数，每个元素的值只能取 0、1 或 2。设计一个算法，将所有元素按 0、1、2 的顺序排列。

解：用 $0\sim i$ 表示 0 元素区间，$k\sim n-1$ 表示 2 元素区间，中间部分为 1 元素区间，如图 2.2 所示。初始时，$i=-1$，$k=n$ 表示这些区间为空。用 j 扫描顺序表 L 中部的所有元素，j 的初始值为 0，当 j 所指的元素为 0 时，说明它一定属于前部，i 增 1(扩大 0 元素区间)，将该元素交换到位置 i(从前面交换过来的元素一定是 1)，j 前进；当 j 所指的元素为 2 时，说明它一定属于后部，k 减 1(扩大 2 元素区间)，将该元素交换到位置 k，若此时 j 前进则会导致该位置不能被交换到前部，所以 j 不前进；当 j 所指的元素为 1 时，说明它一定属于中部，保持原来的位置不动，j 前进。对应的算法如下：

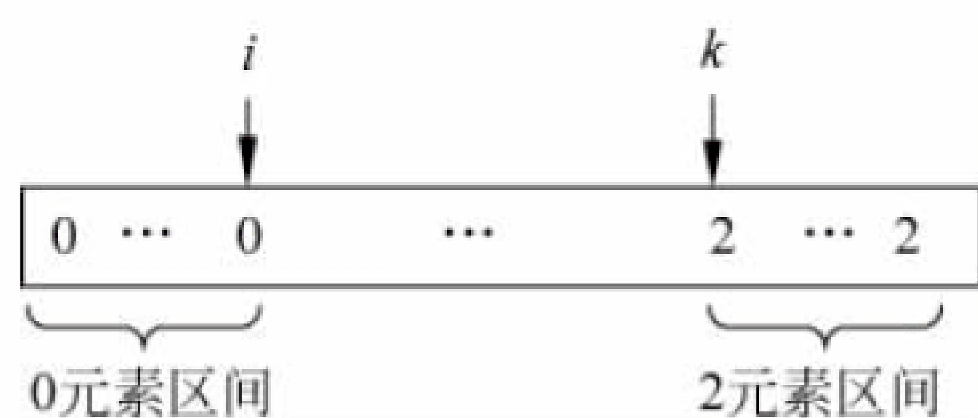


图 2.2 3 个区间

```
void move(SqList *&L)
{   int i=-1,j=0,k=L->length;
    while (j<k)
    {   if (L->data[j]==0)
        {   i++;
            swap(L->data[i],L->data[j]);
            j++;
        }
        else if (L->data[j]==2)
        {   k--;
            swap(L->data[k],L->data[j]);
        }
        else j++;
    }
}
```

4.【顺序表算法】若一个线性表采用顺序表 L 存储，其中所有元素为整数。设计一个时间和空间两方面尽可能高效的算法将所有元素划分成两部分，其中前半部分的每个元素均小于等于整数 $k1$，后半部分的每个元素均大于等于整数 $k2$。例如，对于(6,4,10,7,9,2,20,1,3,30)，当 $k1=5$，$k2=8$ 时，一种结果为([3,4,1,2],6,7,[20,10,9,30])。如果

$k1>k2$,算法返回 false,否则返回 true。

解:当 $k1\leqslant k2$ 时,先将所有小于等于整数 $k1$ 前移,置 $i=0,j=n-1$,从左向右找大于 $k1$ 的元素 data[i],再从右向左找小于等于 $k1$ 的元素 data[j],将两者交换,如此直到 $i=j$ 为止。然后采用类似的方法将 data[$i..n-1$]中所有大于等于 $k2$ 的元素移到右半部分,最后返回真。如果 $k1>k2$,直接返回假。对应的算法如下:

```
bool Rearrangement(SqList *&L,int k1,int k2)
{   int i=0,j=L->length-1;
    if (k1>k2)                      //参数错误返回假
        return false;
    while (i<j)                     //将所有小于等于 k1 的元素前移
    {   while (L->data[i]<=k1) i++;
        while (L->data[j]>k1) j--;
        if (i<j)
        {   swap(L->data[i],L->data[j]);
            i++; j--;
        }
    }
    j=L->length-1;
    while (i<j)                     //将所有大于等于 k2 的元素后移
    {   while (L->data[i]<k2) i++;
        while (L->data[j]>=k2) j--;
        if (i<j)
        {   swap(L->data[i],L->data[j]);
            i++;j--;
        }
    }
    return true;                    //操作成功返回真
}
```

5.【有序顺序表算法】用顺序表 A 和 B 表示的两个线性表,元素的个数分别为 m 和 n,假设表中元素都是递增排列的,且这($m+n$)个元素中没有重复的。

(1) 设计一个算法将这两个线性表合并成一个递增有序线性表,并存储到另一个顺序表 C 中。

(2) 如果顺序表 A 的大小为($m+n$)个单元,是否可以不利用顺序表 C 而将合并结果存放于顺序表 A 中,若可以,请设计对应的算法;若不可以,请说明理由。

(3) 设顺序表 A 中前 k 个元素有序,后 $n-k$ 个元素有序,试设计一个算法使得整个顺序表有序,要求算法的空间复杂度为 $O(1)$。

解:(1) 采用基本的二路归并方法。用 i 和 j 分别扫描顺序表 A 和 B,比较它们当前的元素,将较小者复制到顺序表 C 中,这一过程循环到其中的一个顺序表扫描完毕为止,将另一个顺序表余下的元素全部复制到顺序表 C 中。对应的算法如下:

```
void merge1(SqList *A, SqList *B, SqList *&C)
{   int i=0,j=0,k=0;
    C=(SqList *)malloc(sizeof(SqList));
```

```
        while (i < A -> length && j < B -> length)
        {   if (A -> data[i] < B -> data[j])
            {   C -> data[k] = A -> data[i];
                i++;k++;
            }
            else
            {   C -> data[k] = B -> data[j];
                j++;k++;
            }
        }
        while (i < A -> length)
        {   C -> data[k] = A -> data[i];
            i++;k++;
        }
        while (j < B -> length)
        {   C -> data[k] = B -> data[j];
            j++;k++;
        }
        C -> length = k;
    }
```

本算法的时间复杂度为 $O(m+n)$、空间复杂度为 $O(1)$。

(2) 可以，设顺序表 A、B 的长度分别为 m、n，重新置 A 的长度为 $m+n$。$i=0$，$j=0$，$k=m+n-1$，采用二路归并，将归并的元素放在新表 A 中，但从 A 的尾部开始起放置。对应的算法如下：

```
    void merge2(SqList *&A, SqList * B)
    {   int i = A -> length - 1, j = B -> length - 1;
        int k = A -> length + B -> length - 1;
        A -> length = A -> length + B -> length;  //重置 A 的长度
        while (i >= 0 && j >= 0)                  //原来的 A 和 B 都没有扫描完毕时
        {   if (A -> data[i] >= B -> data[j])     //将较大元素 A -> data[i]复制到新表 A 的尾部
            {   A -> data[k] = A -> data[i];
                i--; k--;
            }
            else                                  //将较小元素 B -> data[j]复制到新表 A 的尾部
            {   A -> data[k] = B -> data[j];
                j--; k--;
            }
        }
        while (j >= 0)                            //将 B 的余下元素复制到新表 A 的尾部
        {   A -> data[k] = B -> data[j];
            j--; k--;
        }
    }
```

本算法的时间复杂度为 $O(m+n)$，空间复杂度为 $O(1)$。

(3) 将顺序表 A 的后半部分插入到前半部分中，使整个表有序。用 j 扫描后半部分的有序表，用 i 记录在前半部分有序表中要插入 $A->$data[j]元素的位置。对应的算法如下：

```
void merge3(SqList *&A, int k)
{   int i = 0, i1, j = k;              //j 扫描后半部分的有序表,同时记录前半部分有序表的长度
    ElemType tmp;
    while (j < A -> length && i < j)
    {   if (A -> data[j]> A -> data[j - 1])      //整个表已递增有序,退出循环
            break;
        else if (A -> data[j]< A -> data[i])     //将 A -> data[j]插入到前半部分中
        {   tmp = A -> data[j];
            for (i1 = j - 1; i1 >= i; i1 -- )     //将 A -> data[i]及之后的元素后移
                A -> data[i1 + 1] = A -> data[i1];
            A -> data[i] = tmp;                   //将 A -> data[j]插入到 A -> data[i]处
            i++; j++;
        }
        else i++;
    }
}
```

本算法的时间复杂度为 $O(m \times n)$、空间复杂度为 $O(1)$。

6.【有序顺序表算法】假设表示集合的顺序表是一个有序顺序表,设计一个高效的算法实现集合的求交集运算,即 $C=A \cap B$。

解:可以利用有序表二路归并方法来提高算法效率,在归并过程中只将有序顺序表 A、B 中公共的元素复制到 C 中。对应的算法如下:

```
void Intersection(SqList *A, SqList *B, SqList *&C)
{   int i = 0, j = 0, k = 0;                      //k 记录 C 中的元素个数
    C = (SqList *)malloc(sizeof(SqList));
    while (i < A -> length && j < B -> length)
    {   if (A -> data[i] == B -> data[j])         //将公共的元素放入 C 中
        {   C -> data[k] = A -> data[i];
            i++; j++; k++;
        }
        else if (A -> data[i]< B -> data[j]) i++;
        else j++;
    }
    C -> length = k;                              //修改集合的长度
}
```

本算法的时间复杂度为 $O(m+n)$、空间复杂度为 $O(1)$,其中 m、n 分别为两个顺序表的长度。

7.【有序顺序表算法】假设表示集合的顺序表是一个有序顺序表,设计一个高效的算法实现集合的求并集运算,即 $C=A \cup B$。

解:可以利用有序表二路归并方法来提高算法效率。在归并过程中将有序顺序表 A、B 中不同的元素复制到 C 中,公共的元素只复制一次,当有一个表扫描完毕时将另外一个表的所有元素复制到 C 中。对应的算法如下:

```
void Union(SqList *A, SqList *B, SqList *&C)
{   int i = 0, j = 0, k = 0;                      //k 记录 C 中的元素个数
```

```
    C = (SqList *)malloc(sizeof(SqList));
    while (i < A -> length && j < B -> length)
    {   if (A -> data[i]< B -> data[j])
        {   C -> data[k] = A -> data[i];
            i++; k++;
        }
        else if (A -> data[i]> B -> data[j])
        {   C -> data[k] = B -> data[j];
            j++; k++;
        }
        else                            //公共元素只复制一次
        {   C -> data[k] = A -> data[i];
            i++; j++; k++;
        }
    }
    while (i < A -> length)             //若 A 未遍历完,将余下的所有元素复制到 C 中
    {   C -> data[k] = A -> data[i];
        i++; k++;
    }
    while (j < B -> length)             //若 B 未遍历完,将余下的所有元素复制到 C 中
    {   C -> data[k] = B -> data[j];
        j++; k++;
    }
    C -> length = k;                    //修改顺序表的长度
}
```

本算法的时间复杂度为 $O(m+n)$、空间复杂度为 $O(1)$，其中 m、n 分别为两个顺序表的长度。

8.【有序顺序表算法】假设表示集合的顺序表是一个有序顺序表，设计一个高效的算法实现集合的求差集运算，即 $C=A-B$。

解：可以利用有序表二路归并方法来提高算法效率。在归并过程中只将有序顺序表 A 中较小的元素复制到 C 中，若 B 表扫描完毕，表示 A 表余下的元素都较大，将它们均复制到 C 中。对应的算法如下：

```
void Diffence(SqList *A, SqList *B, SqList *&C)
{   int i = 0, j = 0, k = 0;                //k 记录 C 中的元素个数
    C = (SqList *)malloc(sizeof(SqList));
    while (i < A -> length && j < B -> length)
    {   if (A -> data[i]< B -> data[j])     //只将 A 中小的元素放入 C 中
        {   C -> data[k] = A -> data[i];
            i++; k++;
        }
        else if (A -> data[i]> B -> data[j])
            j++;
        else                                //公共元素不能放入 C 中
        {   i++;
            j++;
```

```
            }
        }
        while (i < A -> length)                //若 A 未遍历完,将余下的所有元素放入 C 中
        {   C -> data[k] = A -> data[i];
            i++; k++;
        }
        C -> length = k;                       //修改集合的长度
    }
```

本算法的时间复杂度为 $O(m+n)$、空间复杂度为 $O(1)$，其中 m、n 分别为两个顺序表的长度，和上例相比，因为数据有序而降低了算法的时间复杂度。

9.【单链表算法】设计一个算法，查找带头结点的非空单链表 L 中的最后一个最小结点（最小结点可能有多个），并返回该结点的逻辑序号。

解：通过遍历方法查找最后一个最小结点 minp，用 mini 记录其逻辑序号。对应的算法如下：

```
int MinLast(LinkNode * L)
{   LinkNode * p = L -> next, * minp = p;
    int i = 1, mini = i;
    while (p!= NULL)
    {   if (minp -> data >= p -> data)
        {   minp = p;
            mini = i;
        }
        i++;
        p = p -> next;
    }
    return mini;
}
```

10.【单链表算法】有两个整数序列 $A=(a_1,a_2,\cdots,a_n)$ 和 $B=(b_1,b_2,\cdots,b_m)$，分别用单链表 ha 和 hb 存储，设计一个算法判断 B 是否为 A 的连续子序列。

解：采用穷举法进行判断，即从 ha 的每一个结点开始与 hb 的所有结点依次匹配，若 hb 比较完毕，表示是子序列，返回真；如果 ha 的所有结点都比较完毕，表示不是子序列，返回假。对应的算法如下：

```
bool subseq(LinkNode * ha, LinkNode * hb)
{   LinkNode * pa = ha -> next, * pb, * pa1, * pb1;
    while (pa!= NULL)
    {   pb = hb -> next;                        //pb 指向 hb 的首结点
        pa1 = pa; pb1 = pb;
        while (pa1!= NULL && pb1!= NULL && pa1 -> data == pb1 -> data)
        {   pa1 = pa1 -> next;                  //若相等,继续比较后续结点
            pb1 = pb1 -> next;
        }
        if (pb1 == NULL)                        //匹配成功返回 true
            return true;
```

```
            pa = pa -> next;                        //从 ha 的下一个结点继续匹配
        }
        return false;
    }
```

11.【单链表算法】设计一个算法判定单链表 L(带头结点)是否为递增的。

解：从链表 L 的第 2 个结点开始，判定每个结点的值是否比其前驱结点的值大。若有一个不成立，则整个链表便不是递增的，否则是递增的。对应的算法如下：

```
bool increase(LinkNode * L)
{   LinkNode * p = L -> next, * post;           //p 指向首结点
    post = p -> next;                            //post 指向 p 结点的后继结点
    while (post!= NULL)
    {   if (post -> data > p -> data)            //若正序则继续判断下一个结点
        {   p = post;                            //p、post 同步后移
            post = post -> next;
        }
        else
            return false;
    }
    return true;
}
```

12.【单链表算法】假定采用带头结点的单链表保存单词，当两个单词有相同的后缀时可共享相同的后缀存储空间，例如“loading”和“being”，如图 2.3 所示。设 str1 和 str2 分别指向两个单词所在单链表的头结点，链表结点结构为(data，next)。请设计一个时间上尽可能高效的算法，找出由 str1 和 str2 所指向两个链表共同后缀的起始位置(如图中字符'i'所在结点的位置 p)。

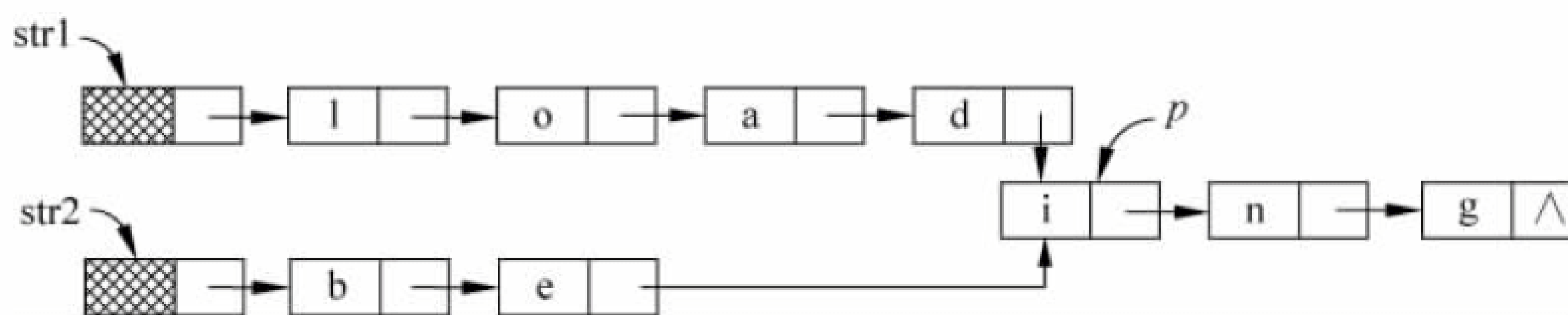


图 2.3 两个单词的后缀共享存储结构

解：分别求出 str1 和 str2 所指的两个链表的长度 m 和 n。将两个链表以表尾对齐，令指针 p、q 分别指向 str1 和 str2 的头结点，若 $m \geqslant n$，则让 p 指向 str1 链表中的第 $m-n+1$ 个结点；若 $m<n$，则让 q 指向 str2 链表中的第 $n-m+1$ 个结点，即让指针 p、q 所指结点到表尾的长度相等。反复将指针 p、q 同步后移，并判断它们是否指向同一结点。若 p、q 指向同一结点，则该结点即为所求的共同后缀的起始位置。对应的算法如下：

```
LinkNode * Findlist(LinkNode * str1,LinkNode * str2)
{   int m,n;
    LinkNode * p, * q;
```

```
        m = ListLength(str1);                  //求单链表 str1 的长度 m
        n = ListLength(str2);                  //求单链表 str2 的长度 n
        for (p = str1;m > n;m -- )             //若 m 大,则 p 后移到 str1 的第 m - n + 1 个结点
            p = p -> next;
        for (q = str2;m < n;n -- )             //若 n 大,则 q 后移到 str2 的第 m - n + 1 个结点
            q = q -> next;
        while (p -> next!= NULL && p -> next!= q -> next)
        {   p = p -> next;                     //p、q 两步后移找第一个指针值相等的结点
            q = q -> next;
        }
        return p -> next;
    }
```

13.【单链表算法】某非空单链表 L 中的所有元素为整数,设计一个算法将所有小于零的结点移到所有大于等于零的结点的前面。

解:用 p 指针扫描单链表 L,pre 指向其前驱结点。当 p 不空时循环,若 p 所指结点的 data 值小于 0,通过 pre 指针将其从链表中移去,然后将 p 结点插入到表头,并置 p=pre->next;否则,pre、p 同步后移一个结点。对应的算法如下:

```
    void Move(LinkNode *&L)
    {   LinkNode *p = L -> next, *pre = p;
        while (p!= NULL)
        {   if (p -> data < 0)
            {   pre -> next = p -> next;          //将结点 p 从链表中移去
                p -> next = L -> next;            //将结点 p 插入到表头
                L -> next = p;
                p = pre -> next;
            }
            else
            {   pre = p;                          //pre、p 同步后移
                p = p -> next;
            }
        }
    }
```

14.【单链表算法】设计一个算法删除带头结点的非空单链表 L 中第一个值为 x 的结点(值为 x 的结点可能有多个),成功操作返回 true,否则返回 false。

解:用 pre、p 遍历整个单链表,pre 指向结点 p 的前驱结点,p 用于查找第一个值为 x 的结点,当找到后通过 pre 结点将 p 结点删除,返回真,否则返回假。对应的算法如下:

```
    bool DelFirstx(LinkNode *&L,ElemType x)
    {   LinkNode *pre = L, *p = pre -> next;              //pre 指向结点 p 的前驱结点
        while (p!= NULL && p -> data!= x)
        {   pre = p;
            p = p -> next;                                 //pre、p 同步后移
        }
        if (p!= NULL)                                      //找到值为 x 的 p 结点
```

```
        {   pre->next=p->next;
            free(p);
            return true;
        }
        else return false;                              //未找到值为x的结点
    }
```

15.【单链表算法】有一个带头结点的非空单链表 L，设计一个算法，删除其中第1、3、5、…的结点，即删除奇数序号的结点，并讨论算法的复杂度。

解：用 p 遍历单链表中奇数序号的结点，pre 指向其前驱结点。在 p 不为空时循环，即通过 pre 结点将 p 结点删除。对应的算法如下：

```
void Delodd(LinkNode *&L)
{   LinkNode *pre=L,*p=pre->next;
    while (p!=NULL)
    {   pre->next=p->next;                              //p指向奇数序号的结点
        free(p);                                        //删除并释放p结点
        pre=pre->next;                                  //pre指向偶数序号的结点
        if (pre==NULL) break;                           //当pre为空时表示所有结点扫描完毕
        p=pre->next;
    }
}
```

16.【单链表算法】设计一个算法，删除带头结点的单链表 L 中 data 值大于或等于 min、小于或等于 max 之间的结点(若表中有这样的结点)，同时释放被删结点的空间，这里 min 和 max 是两个给定的参数，并分析算法的时间复杂度。

解：用 p 遍历单链表 L，pre 指向它的前驱结点，若 p 结点满足被删条件，则通过 pre 结点删除之并将 p 下移一个结点，否则 pre、p 同步后移一个结点。对应的算法如下：

```
void Delnodes1(LinkNode *&L,ElemType min,ElemType max)
{   LinkNode *pre=L,*p=pre->next;
    while (p!=NULL)
    {   if (p->data>=min && p->data<=max)               //p结点为被删结点,pre为前驱结点
        {   pre->next=p->next;                          //删除p结点
            free(p);                                    //释放p结点的空间
            p=pre->next;                                //p下移一个结点
        }
        else
        {   pre=p;
            p=p->next;                                  //pre、p同步后移一个结点
        }
    }
}
```

17.【单链表算法】有一个由整数元素构成的非空单链表 A，设计一个算法，将其拆分成两个单链表 A 和 B，使得 A 单链表中含有所有的偶数结点，B 单链表中含有所有的奇数结点，且保持原来的相对次序。

解：采用尾插法建立新表 A、B。用 p 遍历原单链表 A 中的所有数据结点，若为偶数结点，则将其链到 A 中；若为奇数结点，则将其链到 B 中。对应的算法如下：

```
void Split(LinkNode *&A,LinkNode *&B)
{   LinkNode *p=A->next, *ra, *rb;
    ra=A;
    B=(LinkNode *)malloc(sizeof(LinkNode));        //建立头结点 B
    rb=B;                                          //rb 总是指向 B 链表的尾结点
    while (p!=NULL)
    {   if (p->data%2==0)                          //若为偶数结点
        {   ra->next=p;                            //将 p 结点链到 A 中
            ra=p;
            p=p->next;
        }
        else                                       //若为奇数结点
        {   rb->next=p;                            //将 p 结点链到 B 中
            rb=p;
            p=p->next;
        }
    }
    ra->next=rb->next=NULL;                        //尾结点的 next 域置为空
}
```

18.【单链表算法】已知由单链表 L 表示的线性表中含有 3 类字符的数据元素(如字母字符、数字字符和其他字符)。设计一个算法构造 3 个循环单链表 A、B、C，使每个循环单链表中只含同一类的字符，且利用原表中的结点空间作为这 3 个表的结点空间，头结点可另辟空间。

解：先创建 3 个空的循环单链表，用 p 扫描单键表 L 的所有数据结点，将不同类型的结点采用头插法插入到相应的循环单链表中。对应的算法如下：

```
void Split(LinkNode *L,LinkNode *&A,LinkNode *&B,LinkNode *&C)
{   LinkNode *p=L->next, *q;
    A=L;
    A->next=A;                                 //A 为空的循环单链表
    B=(LinkNode *)malloc(sizeof(LinkNode));    //创建 B 的头结点
    B->next=B;                                 //B 为空的循环单链表
    C=(LinkNode *)malloc(sizeof(LinkNode));    //创建 C 的头结点
    C->next=C;                                 //C 为空的循环单链表
    while (p!=NULL)                            //扫描原来 A 的所有数据结点
    {   if (p->data>='A' && p->data<='Z' || p->data>='a' && p->data<='z')
        {   q=p;p=p->next;
            q->next=A->next;A->next=q;         //结点 q 采用头插法插入到 A 中
        }
        else if (p->data>='0' && p->data<='9')
        {   q=p;p=p->next;
            q->next=A->next;A->next=q;         //结点 q 采用头插法插入到 B 中
        }
```

```
            else
            {   q = p;p = p -> next;
                q -> next = C -> next;C -> next = q;     //结点 q 采用头插法插入到 C 中           }
        }
    }
```

19.【有序单链表算法】有一个递增单链表 L(允许出现值域重复的结点),设计一个算法删除值域重复的结点。

解:由于是有序表,所以相同值域的结点都是相邻的。用 p 扫描递增单链表,若 p 结点的值域等于其后结点的值域,则删除后者。对应的算法如下:

```
void Dels(LinkNode *&L)
{   LinkNode *p = L -> next, *q;
    while (p -> next!= NULL)
    {   if (p -> data == p -> next -> data)          //找到重复值的结点
        {   q = p -> next;                           //q 指向这个重复值的结点
            p -> next = q -> next;                   //删除 q 结点
            free(q);
        }
        else p = p -> next;
    }
}
```

本算法的时间复杂度为 $O(n)$。

20.【有序单链表算法】设 A 和 B 是两个带头结点的单链表,其表中元素递增有序。试写一算法将 A 和 B 归并成一个按元素值递减有序的单链表 C,并要求辅助空间为 $O(1)$,请分析算法的时间复杂度。

解:由于要求辅助空间为 $O(1)$,所以需要就地处理。利用二路归并方法,用 p 扫描 A、q 扫描 B,比较结点 p 和结点 q 的值域大小,将较小者采用头插法插入到 C 中,最后将 A 或 B 中余下的结点采用头插法插入到 C 中。对应的算法如下:

```
void merge(LinkNode *A,LinkNode *B,LinkNode *&C)
{   LinkNode *p = A -> next, *q = B -> next, *r;
    free(B);
    C = A;
    C -> next = NULL;                        //新建立一个空的单链表 C
    while (p!= NULL && q!= NULL)
    {   if (p -> data < q -> data)           //将 p 结点采用头插法插入到 C 中
        {   r = p -> next;
            p -> next = C -> next;
            C -> next = p;
            p = r;
        }
        else                                 //将 q 结点采用头插法插入到 C 中
        {   r = q -> next;
            q -> next = C -> next;
```

```
            C->next=q;
            q=r;
        }
    }
    if (q!=NULL) p=q;
    while (p!=NULL)                         //将余下的结点采用头插法插入到C中
    {   r=p->next;
        p->next=C->next;
        C->next=p;
        p=r;
    }
}
```

上述算法的时间复杂度为$O(m+n)$，m和n分别为A、B的数据结点个数。

21.【有序单链表算法】已知单链表L(带头结点)是一个递增有序表，试写一高效算法删除表中值大于min且小于max的结点(若表中有这样的结点)，同时释放被删结点的空间，这里min和max是两个给定的参数。请分析你的算法时间复杂度。

解：由于单链表L是一个递增有序表，首先找到值域刚好大于min的结点p，再依次删除所有值域小于max的结点。对应的算法如下：

```
void Delnodes(LinkNode *&L,ElemType min,ElemType max)
{   LinkNode *p=L->next, *pre=L;            //pre指向结点p的前驱结点
    while (p!=NULL && p->data<=min)         //查找刚好大于min的结点p
    {   pre=p;                              //pre、p同步后移
        p=p->next;
    }
    while (p!=NULL && p->data<max)          //删除所有小于max的结点
    {   pre->next=p->next;
        free(p);
        p=pre->next;
    }
}
```

本算法的时间复杂度为$O(n)$。

22.【双链表算法】有一个非空双链表L，设计一个算法在第i个结点之后插入一个值为x的结点。

解：先在L中查找第i个结点p，若不存在这样的结点返回false；否则新建一个值为x的结点s，在结点p之后插入s结点，返回true。对应的算法如下：

```
bool InsertAfteri(DLinkNode *&L,int i,ElemType x)
{   DLinkNode *p=L->next, *s;
    int j=1;
    if (i<1) return false;
    while (p!=NULL && j<i)              //查找第i个结点p
    {   j++;
        p=p->next;
    }
```

```
    if (p == NULL)                    //没有找到第 i 个结点返回假
        return false;
    s = (DLinkNode *)malloc(sizeof(DLinkNode));
    s->data = x;                      //在 p 结点之后插入新结点
    if (p->next!= NULL) p->next->prior = s;
    s->next = p->next;
    p->next = s;
    s->prior = p;
    return true;
}
```

23.【双链表算法】有一个非空双链表 L,设计一个算法删除所有值为 x 的结点。

解：先让 p 指向首结点。p 不为空时循环：若 p 所指结点值为 x，删除 p 结点，让 p 指向下一个结点；否则 p 后移一个结点。对应的算法如下：

```
void DeleteAllx(DLinkNode *&L,ElemType x)
{   DLinkNode *p = L->next, *q;
    while (p!= NULL)
    {   if (p->data == x)
        {   q = p->next;              //临时保存 p 结点的后继结点
            p->prior->next = p->next;
            if (p->next!= NULL)
                p->next->prior = p->prior;
            p = q;
        }
        else p = p->next;
    }
}
```

24.【双链表算法】有一个带头结点的双链表 L,其所有元素均为整数,设计一个算法删除所有奇数元素的结点。

解：先让 p 指向首结点。p 不为空时循环：若 p 所指结点值为奇数，让 q 指向其后继结点，删除 p 结点，置 p 为 q；否则 p 后移一个结点。对应的算法如下：

```
void DelOdd(DLinkNode *&L)
{   DLinkNode *p = L->next, *q;
    while (p!= NULL)
    {   if (p->data % 2 == 1)             //p 指向奇数结点
        {   q = p->next;
            p->prior->next = q;
            q->prior = p->prior;
            free(p);
            p = q;
        }
        else p = p->next;                 //p 指向偶数结点
    }
}
```

25.【循环单链表算法】已知带头结点的循环单链表 L 中至少有两个结点，每个结点的两个域为 data 和 next，其中 data 的类型为整型。设计一个算法判断该链表中每个结点值是否小于其后续两个结点值之和，若满足，返回 true，否则返回 false。

解：用 p 扫描整个循环单链表 L，一旦找到 p—> data < p—> next—> data+p—> next—> next—> data 条件不成立的结点 p，则中止循环，返回假，否则继续扫描。当 while 循环正常结束时返回真。对应的算法如下：

```
bool Judge(LinkNode * L)
{    LinkNode * p = L -> next;
     boolflag = true;
     while (p -> next -> next!= L &&flag)
     {    if (p -> data > p -> next -> data + p -> next -> next -> data)
               flag = false;
          p = p -> next;
     }
     return flag;
}
```

26.【循环单链表算法】设计一个算法在带头结点的非空循环单链表 L 中第一个最大值结点(最大值结点可能有多个)之前插入一个值为 x 的结点。

解：用 p 扫描单链表 L，pre 指向 p 结点的前驱结点，maxp 指向最大值结点，maxpre 指向最大值结点的前驱结点。当 p 不为 L 时循环：若 p 结点值大于 maxp 结点值，置 maxpre 为 pre、maxp 为 p。然后 pre、p 同步后移一个结点。最后新建一个存放 x 的结点 s，将其插入到 maxpre 结点之后。对应的算法如下：

```
void Insertbeforex(LinkNode * &L, ElemType x)
{    LinkNode * p = L -> next, * pre = L;
     LinkNode * maxp = p, * maxpre = L, * s;
     while (p!= L)
     {    if (maxp -> data < p -> data)
          {    maxp = p;
               maxpre = pre;
          }
          pre = p;
          p = p -> next;
     }
     s = (LinkNode * )malloc(sizeof(LinkNode));
     s -> data = x;
     s -> next = maxpre -> next;
     maxpre -> next = s;
}
```

27.【循环双链表算法】有一个非空循环双链表 L，设计一个算法删除第一个值为 x 的结点。

解：让 p 指向首结点。p 不为 L 且 p 结点值不为 x 时循环：p 后移一个结点。循环结束时，若 p 为 L 表示未找到值为 x 的结点，返回 false，否则删除 p 结点并返回 true。对应的

算法如下：

```
bool DeleteFirstx(DLinkNode *&L, ElemType x)
{   DLinkNode *p = L->next;
    while (p!= L && p->data!= x)
        p = p->next;
    if (p == L)                        //未找到值为x的结点返回false
        return false;
    else
    {   p->prior->next = p->next;      //删除p结点
        p->next->prior = p->prior;
        free(p);
        return true;
    }
}
```

第3章 栈和队列



3.1 本章知识体系

1. 知识结构图

本章的知识结构如图 3.1 所示。

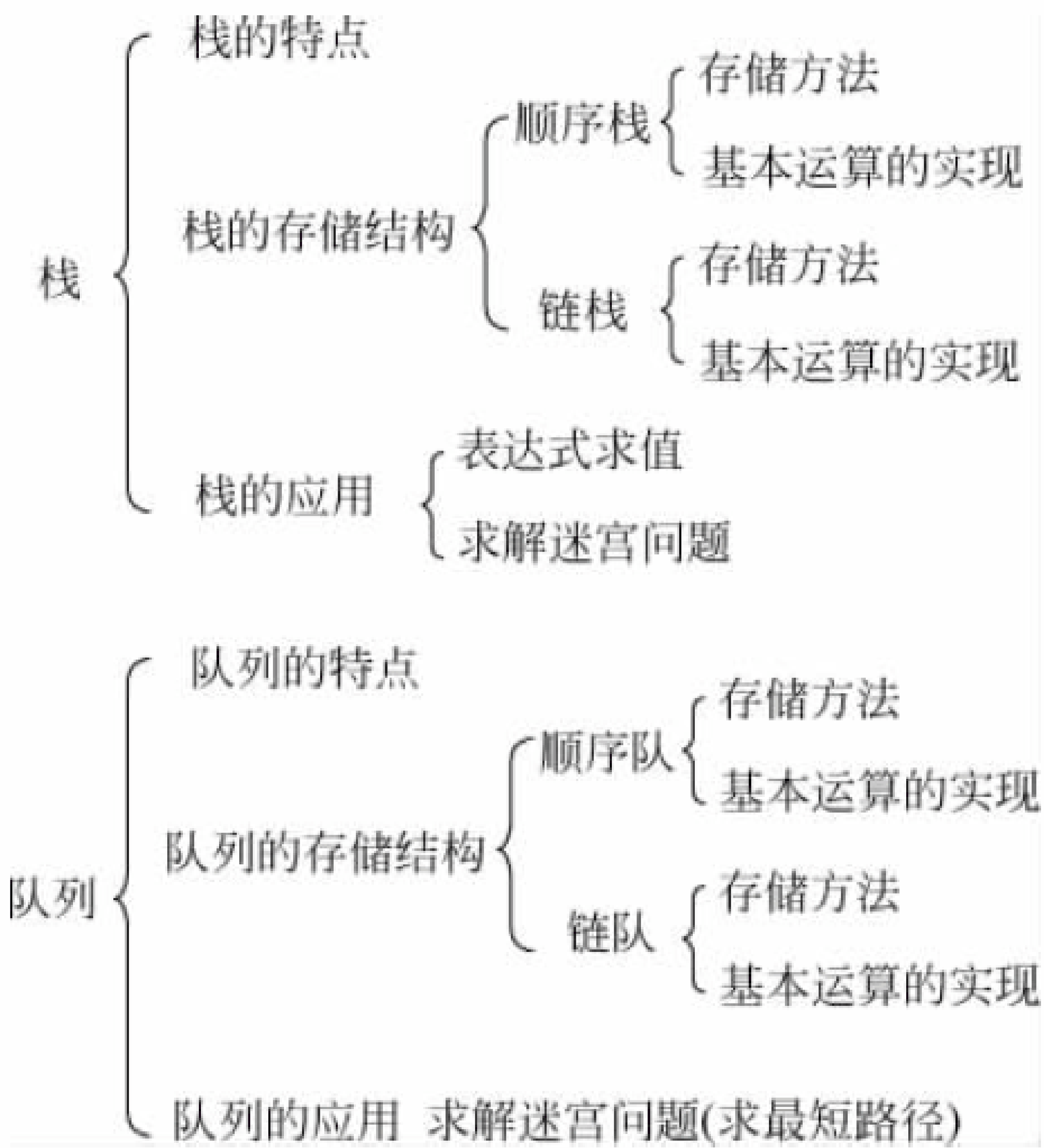


图 3.1 第 3 章知识结构图

2. 基本知识点

（1）栈、队列和线性表的异同。

（2）顺序栈的基本运算算法设计。

（3）链栈的基本运算算法设计。

（4）顺序队的基本运算算法设计。

（5）环形队列和非环形队列的特点。

（6）链队的基本运算算法设计。

（7）利用栈/队列求解复杂的应用问题。

3. 要点归纳

（1）栈和队列的共同点是它们的数据元素都呈线性关系，且只允许在端点处插入和删除元素。

（2）栈是一种“后进先出”的数据结构，只能在同一端进行元素的插入和删除。

（3）栈可以采用顺序栈和链栈两类存储结构。

（4）n 个不同元素的进栈顺序和出栈顺序不一定相同。

（5）在顺序栈中通常用栈顶指针指向当前栈顶的元素。

（6）在顺序栈中用数组 data[0..MaxSize－1]存放栈中元素，只能将一端作为栈底，另一端作为栈顶，通常的做法是将 data[0]端作为栈底，data[MaxSize－1]端作为栈顶。用户

也可以将 data[MaxSize－1]端作为栈底，data[0]端作为栈顶，但不能将中间位置作为栈底或者栈顶。

(7) 初始时栈顶指针 top 设置为－1，栈空的条件为 top＝－1，栈满的条件为 top＝MaxSize－1，元素 x 的进栈操作是 top＋＋；data[top]＝x，出栈操作是 x＝data[top]；top－－。这是经典做法，但不是唯一的方法，如果初始时 top 设置为 0，可以设置栈空的条件为 top＝0，栈满的条件为 top＝MaxSize，元素 x 的进栈操作是 data[top]＝x；top＋＋，出栈操作是 top－－；x＝data[top]。

(8) 在顺序栈或链栈中，进栈和出栈操作不涉及栈中元素的移动。

(9) 在链栈中，由于每个结点是单独分配的，通常不考虑上溢出问题。

(10) 无论是顺序栈还是链栈，进栈和出栈运算的时间复杂度均为 $O(1)$。

(11) 队列是一种“先进先出”的数据结构，只能从一端插入元素，从另一端删除元素。

(12) 队列可以采用顺序队和链队两类存储结构。

(13) n 个元素进队的顺序和出队顺序总是一致的。

(14) 在顺序队中的元素个数可以由队头指针和队尾指针计算出来。

(15) 环形队列也是一种顺序队，是通过逻辑方法使其首尾相连的，解决非环形队列的假溢出现象。

(16) 在环形队列中，队头指针 f 指向队头元素的前一个位置，队尾指针 r 指向队尾元素，这是一种经典做法，但不是唯一的方法，也可以让队头指针 f 指向队头元素。

(17) 无论是顺序队还是链队，进队和出队运算的时间复杂度均为 $O(1)$。

(18) 在实际应用中，一般栈和队列都是用来存放临时数据的，如果先保存的元素先处理，应该采用队列；如果后保存的元素先处理，应该采用栈。

3.2 教材中的练习题及参考答案

1. 有 5 个元素，其进栈次序为 A、B、C、D、E，在各种可能的出栈次序中以元素 C、D 最先出栈(即 C 第一个且 D 第二个出栈)的次序有哪几个？

答：要使 C 第一个且 D 第二个出栈，应是 A 进栈，B 进栈，C 进栈，C 出栈，D 进栈，D 出栈，之后可以有以下几种情况：

(1) B 出栈，A 出栈，E 进栈，E 出栈，输出序列为 CDBAE；

(2) B 出栈，E 进栈，E 出栈，A 出栈，输出序列为 CDBEA；

(3) E 进栈，E 出栈，B 出栈，A 出栈，输出序列为 CDEBA。

所以可能的次序有 CDBAE、CDBEA、CDEBA。

2. 在一个算法中需要建立多个栈(假设 3 个栈或以上)时可以选用以下 3 种方案之一，试问这些方案相比各有什么优缺点？

(1) 分别用多个顺序存储空间建立多个独立的顺序栈。

(2) 多个栈共享一个顺序存储空间。

(3) 分别建立多个独立的链栈。

答：(1) 优点是每个栈仅用一个顺序存储空间时操作简单；缺点是分配空间小了容易

产生溢出，分配空间大了容易造成浪费，各栈不能共享空间。

（2）优点是多个栈仅用一个顺序存储空间，充分利用了存储空间，只有在整个存储空间都用完时才会产生溢出；缺点是当一个栈满时要向左、右查询有无空闲单元，如果有，则要移动元素和修改相关的栈底和栈顶指针。当接近栈满时要查询空闲单元、移动元素和修改栈底、栈顶指针，这一过程计算复杂且十分耗时。

（3）优点是多个链栈一般不考虑栈的溢出；缺点是栈中元素要以指针相链接，比顺序存储多占用了存储空间。

3. 在以下几种存储结构中哪个最适合用作链栈？

（1）带头结点的单链表。

（2）不带头结点的循环单链表。

（3）带头结点的双链表。

答：栈中元素之间的逻辑关系属线性关系，可以采用单链表、循环单链表和双链表之一来存储，而栈的主要运算是进栈和出栈。

当采用(1)时，前端作为栈顶，进栈和出栈运算的时间复杂度为 $O(1)$。

当采用(2)时，前端作为栈顶，当进栈和出栈时首结点都发生变化，还需要找到尾结点，通过修改其 next 域使其变为循环单链表，算法的时间复杂度为 $O(n)$。

当采用(3)时，前端作为栈顶，进栈和出栈运算的时间复杂度为 $O(1)$。

但单链表和双链表相比，其存储密度更高，所以本题中最适合用作链栈的是带头结点的单链表。

4. 简述以下算法的功能（假设 ElemType 为 int 类型）。

```
void fun(ElemType a[],int n)
{   int i;ElemType e;
    SqStack *st1,*st2;
    InitStack(st1);
    InitStack(st2);
    for (i=0;i<n;i++)
        if (a[i]%2==1)
            Push(st1,a[i]);
        else
            Push(st2,a[i]);
    i=0;
    while (!StackEmpty(st1))
    {   Pop(st1,e);
        a[i++]=e;
    }
    while (!StackEmpty(st2))
    {   Pop(st2,e);
        a[i++]=e;
    }
    DestroyStack(st1);
    DestroyStack(st2);
}
```

答：算法的执行步骤如下。

(1) 扫描数组 a，将所有奇数进到 st1 栈中，将所有偶数进到 st2 栈中。

(2) 先将 st1 的所有元素(奇数元素)退栈，放到数组 a 中并覆盖原有位置的元素；再将 st2 的所有元素(偶数元素)退栈，放到数组 a 中并覆盖原有位置的元素。

(3) 销毁两个栈 st1 和 st2。

所以本算法的功能是利用两个栈将数组 a 中的所有奇数元素放到所有偶数元素的前面。例如 ElemType a[]={1,2,3,4,5,6}，执行算法后数组 a 改变为{5,3,1,6,4,2}。

5. 简述以下算法的功能(顺序栈的元素类型为 ElemType)。

```
void fun(SqStack *&st,ElemType x)
{   SqStack * tmps;
    ElemType e;
    InitStack(tmps);
    while(!StackEmpty(st))
    {   Pop(st,e);
        if(e!= x) Push(tmps,d);
    }
    while (!StackEmpty(tmps))
    {   Pop(tmps,e);
        Push(st,e);
    }
    DestroyStack(tmps);
}
```

答：算法的执行步骤如下。

(1) 建立一个临时栈 tmps 并初始化。

(2) 退栈 st 中的所有元素，将不为 x 的元素进栈到 tmps 中。

(3) 退栈 tmps 中的所有元素，并进栈到 st 中。

(4) 销毁栈 tmps。

所以本算法的功能是如果栈 st 中存在元素 x，将其从栈中清除。例如，st 栈中从栈底到栈顶为 a、b、c、d、e，执行算法 fun(st,'c')后，st 栈中从栈底到栈顶为 a、b、d、e。

6. 简述以下算法的功能(栈 st 和队列 qu 的元素类型均为 ElemType)。

```
bool fun(SqQueue *&qu,int i)
{   ElemType e;
    int j = 1;
    int n = (qu->rear - qu->front + MaxSize) % MaxSize;
    if (j<1 || j>n) return false;
    for (j = 1;j<=n;j++)
    {   deQueue(qu,e);
        if (j!= i)
            enQueue(qu,e);
    }
    return true;
}
```

答：算法的执行步骤如下。

（1）求出队列 qu 中的元素个数 n，参数 i 错误时返回假。

（2）qu 出队共计 n 次，除了第 i 个出队的元素以外，其他出队的元素立即进队。

（3）返回真。

所以本算法的功能是删除 qu 中从队头开始的第 i 个元素。例如，qu 中从队头到队尾的元素是 a、b、c、d、e，执行算法 fun(qu,2)后，qu 中从队头到队尾的元素改变为 a、c、d、e。

7. 什么是环形队列？采用什么方法实现环形队列？

答：在用数组表示队列时把数组看成是一个环形的，即令数组中的第一个元素紧跟在最末一个单元之后就形成了一个环形队列。环形队列解决了非环形队列中出现的“假溢出”现象。

通常采用逻辑上求余数的方法来实现环形队列，假设数组的大小为 n，当元素下标 i 增 1 时采用 $i=(i+1)\%n$ 来实现。

8. 环形队列一定优于非环形队列吗？在什么情况下使用非环形队列？

答：队列主要用于保存中间数据，而且保存的数据满足先产生先处理的特点。非环形队列可能存在数据假溢出现象，即队列中还有空间，可是队满的条件却成立了，为此改为环形队列，这样克服了假溢出现象。但并不能说环形队列一定优于非环形队列，因为环形队列中出队元素的空间可能被后来进队的元素覆盖，如果算法要求在队列操作结束后利用进队的所有元素实现某种功能，这样环形队列就不适合了，在这种情况下需要使用非环形队列，例如利用非环形队列求解迷宫路径就是这种情况。

9. 假设以 I 和 O 分别表示进栈和出栈操作，栈的初态和终栈均为空，进栈和出栈的操作序列可表示为仅由 I 和 O 组成的序列。

（1）在下面所示的序列中哪些是合法的？

A. IOIIOIOO　　B. IOOIOIIO　　C. IIIOIOIO　　D. IIIOOIOO

（2）通过对（1）的分析，设计一个算法判定所给的操作序列是否合法，若合法返回真，否则返回假（假设被判定的操作序列已存入一维数组中）。

解：（1）选项 A、D 均合法，而选项 B、C 不合法。因为在选项 B 中先进栈一次，立即出栈 3 次，这会造成栈下溢。在选项 C 中共进栈 5 次，出栈 3 次，栈的终态不为空。

（2）本题使用一个链栈来判断操作序列是否合法，其中 str 为存放操作序列的字符数组，n 为该数组的字符个数（这里的 ElemType 类型设定为 char）。对应的算法如下：

```
bool judge(char str[],int n)
{   int i = 0; ElemType x;
    LinkStNode * ls;
    bool flag = true;
    InitStack(ls);
    while (i < n && flag)
    {   if (str[i] == 'I')          //进栈
            Push(ls,str[i]);
        else if (str[i] == 'O')     //出栈
        {   if (StackEmpty(ls))
                flag = false;       //栈空时
            else
```

```
                    Pop(ls,x);
            }
            else
                flag = false;                   //其他值无效
            i++;
        }
        if (!StackEmpty(ls)) flag = false;
        DestroyStack(ls);
        return flag;
    }
```

10. 假设表达式中允许包含圆括号、方括号和大括号3种括号，编写一个算法判断表达式中的括号是否正确配对。

解：设置一个栈st，扫描表达式exp，当遇到'('、'['或'{'时将其进栈；当遇到')'时，若栈顶是'('，则继续处理，否则以不配对返回假；当遇到']'时，若栈顶是'['，则继续处理，否则以不配对返回假；当遇到'}'时，若栈顶是'{'，则继续处理，否则以不配对返回假。在exp扫描完毕后，若栈不空，则以不配对返回假；否则以括号配对返回真。本题的算法如下：

```
bool Match(char exp[],int n)
{   LinkStNode * ls;
    InitStack(ls);
    int i = 0;
    ElemType e;
    bool flag = true;
    while (i < n && flag)
    {   if (exp[i] == '(' || exp[i] == '[' || exp[i] == '{')
            Push(ls,exp[i]);            //遇到'('、'['或'{',将其进栈
        if (exp[i] == ')')              //遇到')',若栈顶是'(',继续处理,否则以不配对返回
        {   if (GetTop(ls,e))
            {   if (e == '(') Pop(ls,e);
                else flag = false;
            }
            else flag = false;
        }
        if (exp[i] == ']')              //遇到']',若栈顶是'[',继续处理,否则以不配对返回
        {   if (GetTop(ls,e))
            {   if (e == '[') Pop(ls,e);
                else flag = false;
            }
            else flag = false;
        }
        if (exp[i] == '}')              //遇到'}',若栈顶是'{',继续处理,否则以不配对返回
        {   if (GetTop(ls,e))
            {   if (e == '{') Pop(ls,e);
                else flag = false;
            }
            else flag = false;
        }
        i++;
```

```
        }
        if (!StackEmpty(ls)) flag = false;       //若栈不空,则不配对
        DestroyStack(ls);
        return flag;
    }
```

11. 设从键盘输入一序列的字符 a_1、a_2、…、a_n。设计一个算法实现这样的功能：若 a_i 为数字字符，a_i 进队；若 a_i 为小写字母，将队首元素出队；若 a_i 为其他字符，表示输入结束。要求使用环形队列。

解：先建立一个环形队列 qu，用 while 循环接收用户的输入，若输入数字字符，将其进队；若输入小写字母，出队一个元素，并输出它；若为其他字符，则退出循环。本题的算法如下：

```
void fun()
{   ElemType a,e;
    SqQueue *qu;                                //定义队列指针
    InitQueue(qu);
    while (true)
    {   printf("输入 a:");
        scanf("%s",&a);
        if (a>='0' && a<='9')                   //为数字字符
        {   if (!enQueue(qu,a))
                printf("  队列满,不能进队\n");
        }
        else if (a>='a' && a<='z')              //为小写字母
        {   if (!deQueue(qu,e))
                printf("  队列空,不能出队\n");
            else
                printf("  出队元素:%c\n",e);
        }
        else break;                             //为其他字符
    }
    DestroyQueue(qu);
}
```

12. 设计一个算法，将一个环形队列（容量为 n，元素下标从 0 到 $n-1$）的元素倒置。例如，图 3.2(a)为倒置前的队列（$n=10$），图 3.2(b)为倒置后的队列。

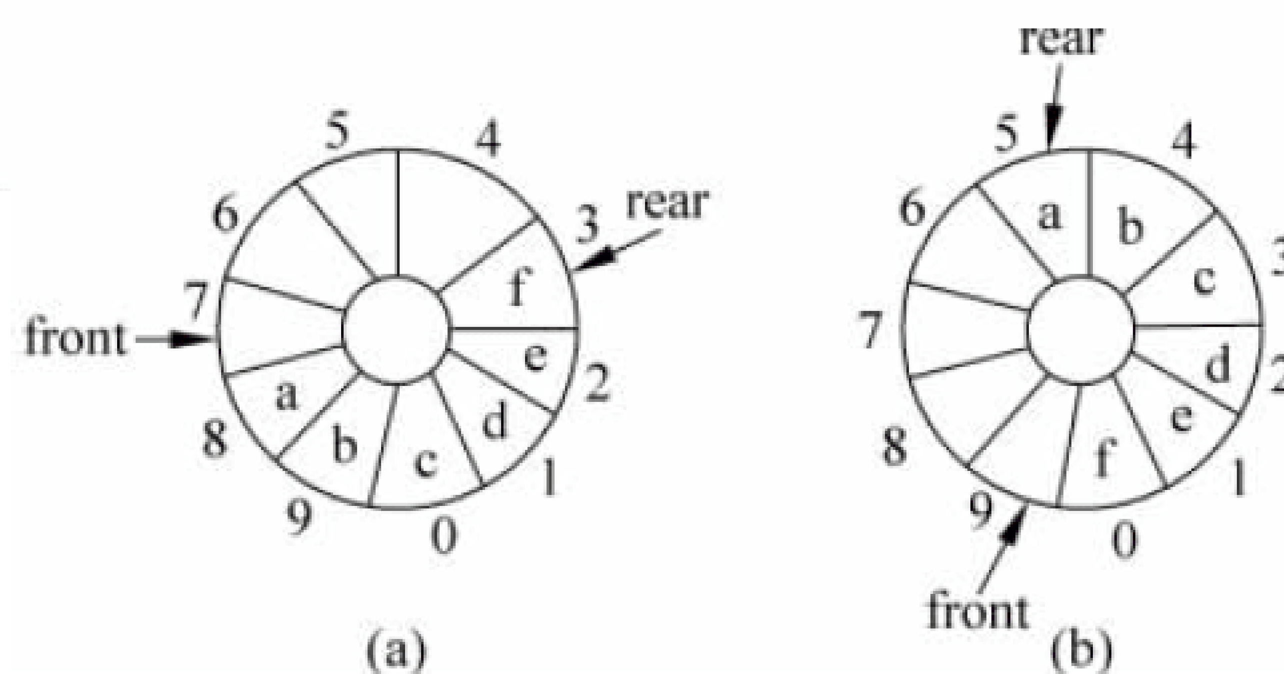


图 3.2　一个环形队列倒置前后的状态

解：使用一个临时栈 st，先将 qu 队列中的所有元素出队并将其进栈 st，直到队列空为止。然后初始化队列 qu(队列清空)，再出栈 st 的所有元素并将其进队 qu，最后销毁栈 st。对应的算法如下：

```
void Reverse(SqQueue *&qu)
{   ElemType e;
    SqStack *st;
    InitStack(st);
    while (!QueueEmpty(qu))            //队不空时出队并进栈
    {   deQueue(qu,e);
        Push(st,e);
    }
    InitQueue(qu);                     //队列初始化
    while (!StackEmpty(st))            //栈不空时出栈并将元素入队
    {   Pop(st,e);
        enQueue(qu,e);
    }
    DestroyStack(st);
}
```

13. 编写一个程序，输入 n(由用户输入)个 10 以内的数，每输入 $i(0\leqslant i\leqslant 9)$就把它插入到第 i 号队列中，最后把 10 个队中的非空队列按队列号从小到大的顺序串接成一条链，并输出该链的所有元素。

解：建立一个队头指针数组 quh 和队尾指针数组 qut，quh[i]和 qut[i]表示 i 号$(0\leqslant i\leqslant 9)$队列的队头和队尾，先将它们的所有元素置为 NULL。对于输入的 x，采用尾插法将其链到 x 号队列中。然后按 0～9 编号的顺序把这些队列中的结点构成一个不带头结点的单链表，其首结点指针为 head。最后输出单链表 head 的所有结点值并释放所有结点。对应的程序如下：

```
#include <stdio.h>
#include <malloc.h>
#define MAXQNode 10                              //队列的个数
typedef struct node
{   int data;
    struct node *next;
} QNode;
void Insert(QNode *quh[],QNode *qut[],int x)    //将 x 插入到相应队列中
{   QNode *s;
    s=(QNode *)malloc(sizeof(QNode));            //创建一个结点 s
    s->data=x; s->next=NULL;
    if (quh[x]==NULL)                            //x 号队列为空队时
    {   quh[x]=s;
        qut[x]=s;
    }
    else                                         //x 号队列不空队时
    {   qut[x]->next=s;                          //将 s 结点链到 qut[x]所指的结点之后
        qut[x]=s;                                //让 qut[x]仍指向尾结点
```

```
        }
    }
    void Create(QNode *quh[],QNode *qut[])          //根据用户的输入创建队列
    {   int n,x,i;
        printf("n:");
        scanf("%d",&n);
        for (i=0;i<n;i++)
        {   do
            {   printf("输入第%d个数:",i+1);
                scanf("%d",&x);
            } while (x<0 || x>10);
            Insert(quh,qut,x);
        }
    }
    void DestroyList(QNode *&head)                   //释放单链表
    {   QNode *pre=head, *p=pre->next;
        while (p!=NULL)
        {
            free(pre);
            pre=p; p=p->next;
        }
        free(pre);
    }
    void DispList(QNode *head)                       //输出单链表的所有结点值
    {   printf("\n输出所有元素:");
        while (head!=NULL)
        {   printf("%d ",head->data);
            head=head->next;
        }
        printf("\n");
    }
    QNode *Link(QNode *quh[],QNode *qut[])           //将非空队列链接起来并输出
    {   QNode *head=NULL, *tail;                     //总链表的首结点指针和尾结点指针
        int i;
        for (i=0;i<MAXQNode;i++)                     //扫描所有队列
            if (quh[i]!=NULL)                        //i号队列不空
            {   if (head==NULL)                      //若i号队列为第一个非空队列
                {   head=quh[i];
                    tail=qut[i];
                }
                else                                 //若i号队列不是第一个非空队列
                {   tail->next=quh[i];
                    tail=qut[i];
                }
            }
        tail->next=NULL;
        return head;
    }
    int main()
    {   int i;
```

```
    QNode * head;
    QNode * quh[MAXQNode], * qut[MAXQNode];        //各队列的队头 quh 和队尾指针 qut
    for (i = 0; i < MAXQNode; i++)
        quh[i] = qut[i] = NULL;                    //置初值空
    Create(quh, qut);                              //建立队列
    head = Link(quh, qut);                         //链接各队列产生单链表
    DispList(head);                                //输出单链表
    DestroyList(head);                             //销毁单链表
    return 1;
}
```

3.3 补充练习题及参考答案



3.3.1 单项选择题

1. 以下数据结构中元素之间为线性关系的是________。

A. 栈　　B. 队列　　C. 线性表　　D. 以上都是

答：D。

2. 栈和队列的共同点是________。

A. 都是先进后出　　B. 都是先进先出

C. 只允许在端点处插入和删除元素　　D. 没有共同点

答：栈和队列都是受限线性表，所谓“受限”指的是在端点处插入和删除元素，所以本题的答案为 C。

3. 经过以下栈运算后 x 的值是________。

InitStack(s)；Push(s,a)；Push(s,b)；Pop(s,x)；GetTop(s,x)；

A. a　　B. b　　C. 1　　D. 0

答：A。

4. 经过以下栈运算后 StackEmpty(s)的值是________。

InitStack(s)；Push(s,a)；Push(s,b)；Pop(s,x)；Pop(s,y)

A. a　　B. b　　C. 1　　D. 0

答：C。

5. 设一个栈的输入序列为 a，b，c，d，则借助一个栈所得到的输出序列不可能是________。

A. a,b,c,d　　B. d,c,b,a　　C. a,c,d,b　　D. d,a,b,c

答：以 d 开头的出栈序列只有 d,c,b,a 一种。本题的答案为 D。

6. 已知一个栈的进栈序列是 1,2,3,…,n，其输出序列是 $p_1,p_2,\cdots,p_n$，若 $p_1=n$，则 p_i 的值是________。

A. i　　B. $n-i$　　C. $n-i+1$　　D. 不确定

答：当 $p_1=n$ 时输出序列只有一种，即 $n,n-1,\cdots,3,2,1$，则 $p_2=n-1,p_3=n-2,\cdots,p_n=1$，推断出 $p_i=n-i+1$，本题的答案为 C。

7. 设 n 个元素进栈序列是 $1,2,3,\cdots,n$，其输出序列是 $p_1,p_2,\cdots,p_n$，若 $p_1=3$，则 p_2 的值________。

A. 一定是 2　　B. 一定是 1　　C. 不可能是 1　　D. 以上都不对

答：当 $p_1=3$ 时，说明 1、2、3 先依次进栈，出栈 3，然后可能出栈 2，也可能是 4 或后面的元素进栈后再出栈。因此，p_2 可能是 2，也可能是 $4,\cdots,n$，但一定不能是 1。本题的答案为 C。

8. 设 n 个元素进栈序列是 $p_1,p_2,p_3,\cdots,p_n$，其输出序列是 $1,2,3,\cdots,n$，若 $p_n=1$，则 $p_i(1\leqslant i\leqslant n-1)$ 的值是________。

A. $n-i+1$　　B. $n-i$　　C. i　　D. 有多种可能

答：当 $p_n=1$ 时，表示它是第一个出栈元素，因此这样的输出序列是唯一的，即有 $p_{n-1}=2,p_{n-2}=3,\cdots,p_1=n$，也就是说 $p_i=n-i+1$。本题的答案为 A。

9. 一个栈的入栈序列为 $1、2、3、\cdots、n$，其出栈序列是 $p_1、p_2、p_3、\cdots、p_n$。若 $p_2=3$，则 p_3 可能取值的个数是________。

A. $n-3$　　B. $n-2$　　C. $n-1$　　D. 无法确定

答：若 1 进栈，1 出栈(p_1)，2 进栈，3 进栈，3 出栈(p_2)，之后可以 2 出栈(p_3)，也可以 $4\sim n$ 的任何元素进栈再出栈(p_3)，所以 p_3 可以是 2 或者 $4\sim n$。另外，1、2 依次进栈，2 出栈(p_1)，3 进栈，3 出栈(p_2)，1 出栈(p_3)。也就是说，p_3 可以是 3 以外的任何元素。本题的答案为 C。

10. 设栈 S 和队列 Q 的初始状态为空，元素 $e_1\sim e_6$ 依次通过栈 S，一个元素出后即进队列 Q，若 6 个元素出队的序列是 $e_2、e_4、e_3、e_6、e_5、e_1$，则栈 S 的容量至少应该是________。

A. 5　　B. 4　　C. 3　　D. 2

答：操作过程为 e_1 进栈，e_2 进栈，e_2 出栈后进队，e_3 进栈，e_4 进栈，e_4 出栈后进队，e_3 出栈后进队，e_5 进栈，e_6 进栈，e_6 出栈后进队，e_5 出栈后进队，e_1 出栈后进队，栈中元素最多时为 3 个。本题的答案为 C。

11. 判定一个顺序栈 st(元素的个数最多为 MaxSize)为空的条件可以设置为________。

A. st->top== MaxSize/2　　B. st->top!= MaxSize/2

C. st->top!=MaxSize-1　　D. st->top==MaxSize-1

答：顺序栈总是以一端(0 或者 MaxSize-1 端)作为栈底，栈空是指栈不存在元素，合适的栈空条件为 st->top== MaxSize-1。本题的答案为 D。

12. 若一个栈用数组 data[1..n]存储，初始栈顶指针 top 为 $n+1$，则以下元素 x 进栈的操作正确的是________。

A. top++;data[top]=x;　　B. data[top]=x;top++;

C. top--;data[top]=x;　　D. data[top]=x;top--;

答：初始栈顶指针 top 为 $n+1$，说明是将 data[n]端作为栈底、data[1]端作为栈顶，在进栈时 top 应递减，由于不存在 data[$n+1$]的元素，所以在进栈时应先将 top 递减，再将 x 放在 top 处。本题的答案为 C。

13. 若一个栈用数组 data[1..n]存储，初始栈顶指针 top 为 n，则以下元素 x 进栈的操

作正确的是________。

A. top++;data[top]=x;　　B. data[top]=x;top++;

C. top--;data[top]=x;　　D. data[top]=x; top--;

答:初始栈顶指针 top 为 n,说明是将 data[n]端作为栈底、data[1]端作为栈顶,在进栈时 top 应递减,由于存在 data[n]的元素,所以在进栈时应先将 x 放在 top 处,再将 top 递减。本题的答案为 D。

14. 若一个栈用数组 data[1..n]存储,初始栈顶指针 top 为 0,则以下元素 x 进栈的操作正确的是________。

A. top++;data[top]=x;　　B. data[top]=x;top++;

C. top--;data[top]=x;　　D. data[top]=x; top--;

答:初始栈顶指针 top 为 0,说明是将 data[1]端作为栈底、data[n]端作为栈顶,在进栈时 top 应递增,由于不存在 data[0]的元素,所以在进栈时应先将 top 递增,再将 x 放在 top 处。本题的答案为 A。

15. 若一个栈用数组 data[1..n]存储,初始栈顶指针 top 为 1,则以下元素 x 进栈的操作正确的是________。

A. top++;data[top]=x;　　B. data[top]=x;top++;

C. top--;data[top]=x;　　D. data[top]=x; top--;

答:初始栈顶指针 top 为 1,说明是将 data[1]端作为栈底、data[n]端作为栈底,在进栈时 top 应递增,由于存在 data[1]的元素,所以在进栈时应先将 x 放在 top 处,再将 top 递增。本题的答案为 B。

说明:从 12～15 小题可以看出,顺序栈的设计并不是唯一的,只要能满足栈的操作特点又能充分利用存储空间就是一种合适的设计。

16. 以下各链表均不带有头结点,其中最不适合用作链栈的链表是________。

A. 只有表头指针没有表尾指针的循环双链表

B. 只有表尾指针没有表头指针的循环双链表

C. 只有表尾指针没有表头指针的循环单链表

D. 只有表头指针没有表尾指针的循环单链表

答:只有表头指针没有表尾指针的循环单链表(不带头结点)在进栈和出栈操作后需要保持循环单链表形式不变,实现进栈和出栈运算的时间复杂度均为 $O(n)$。本题的答案为 D。

17. 由两个栈共享一个数组空间的好处是________。

A. 减少存取时间,降低上溢出发生的几率

B. 节省存储空间,降低上溢出发生的几率

C. 减少存取时间,降低下溢出发生的几率

D. 节省存储空间,降低下溢出发生的几率

答:B。

18. 表达式 a*(b+c)-d 的后缀表达式是________。

A. abcd*+-　　B. abc+*d-

C. abc*+d-　　D. -+*abcd

答：选项 A 对应的中缀表达式为 a－(b＋c * d)，选项 B 对应的中缀表达式为 a *(b＋c)－d，选项 C 对应的中缀表达式为(a＋b * c)－d，选项 D 不是合法的后缀表达式。本题的答案为 B。

19. 在将算术表达式"1＋6/(8－5) * 3"转换成后缀表达式的过程中，当扫描到 5 时运算符栈(从栈顶到栈底次序)为________。

A. － / ＋　　B. － (/ ＋　　C. / ＋　　D. / － ＋

答：算术表达式"1＋6/(8－5) * 3"的后缀表达式是"1 6 8 5 － / 3 * ＋"，当扫描到 5 时，前面的运算符＋、/、(和－均在栈中，运算符栈中从栈顶到栈底次序为－ (/ ＋。本题的答案为 B。

20. 在利用栈求表达式的值时，设立运算数栈 OPND，设 OPND 只有两个存储单元，在求下列表达式中不发生上溢出的是________。

A. a－b * (c＋d)　　B. (a－b) * c＋d

C. (a－b * c)＋d　　D. (a－b) * (c＋d)

答：选项 A 对应的后缀表达式为 a b c d ＋ * －，在求值时 OPND 的最少存储单元为 4。选项 B 对应的后缀表达式为 a b － c * d ＋，在求值时 OPND 的最少存储单元为 2。选项 C 对应的后缀表达式为 a b c * － d ＋，在求值时 OPND 的最少存储单元为 3。选项 D 对应的后缀表达式为 a b － c d ＋ *，在求值时 OPND 的最少存储单元为 3。本题的答案为 B。

21. 经过以下队列运算后 QueueEmpty(qu)的值是________。

InitQueue(qu)；enQueue(qu,a)；enQueue(qu,b)；deQueue(qu,x)；deQueue(qu,y)；

A. a　　B. b　　C. true　　D. false

答：C。

22. 环形队列________。

A. 不会产生下溢出　　B. 不会产生上溢出

C. 不会产生假溢出　　D. 以上都不对

答：C。

23. 在环形队列中元素的排列顺序________。

A. 由元素进队的先后顺序确定　　B. 与元素值的大小有关

C. 与队头和队尾指针的取值有关　　D. 与队中数组大小有关

答：A。

24. 某环形队列的元素类型为 char，队头指针 front 指向队头元素的前一个位置，队尾指针 rear 指向队尾元素，如图 3.3 所示，则队中元素为________。

A. abcd123456　　B. abcd123456c　　C. dfgbca　　D. cdfgbcab

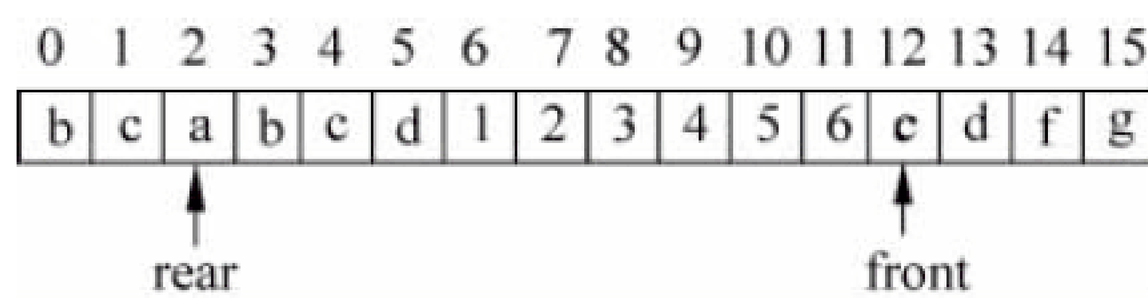


图 3.3　一个环形队列

答：front＝12，队头元素应为下标13的元素，rear＝2，队尾元素应为下标2的元素，队中元素从队头到队尾是data[13..2]，本题的答案为C。

25. 已知环形队列存储在一维数组 $A[0..n-1]$中，且队列非空时front和rear分别指向队头元素和队尾元素。若初始时队列空，且要求第一个进入队列的元素存储在 $A[0]$处，则初始时front和rear的值分别是________。

A. 0，0　　B. 0，$n-1$　　C. $n-1$，0　　D. $n-1$，$n-1$

答：在环形队列中，进队操作是队尾指针rear循环加1，再在该处放置进队的元素，这里要求第一个进队元素存储在 $A[0]$处，则rear应为 $n-1$，因为这样(rear＋1)%n＝0。而队头指向队头元素，此时队头位置为0，所以front的初值为0。本题的答案为B。

26. 若某环形队列有队头指针front和队尾指针rear，在队不满时进队操作仅会改变________。

A. front　　B. rear　　C. front和rear　　D. 以上都不对

答：B。

27. 设环形队列中数组的下标是0～$N-1$，其队头指针为 f(指向队头元素的前一个位置)、队尾指针为 r(指向队尾元素)，则其元素个数为________。

A. $r-f$　　B. $r-f-1$

C. $(r-f)\%N+1$　　D. $(r-f+N)\%N$

答：对于非环形队列，每次是先移动指针，再存取元素，其中的元素个数＝$r-f$，但由于是环形队列，r 可能小于 f，为此求环形队列中元素个数的公式改为 $(r-f+N)\%N$。本题的答案为D。

28. 设环形队列的存储空间为 $a[0..20]$，且当前队头指针(f 指向队首元素的前一个位置)和队尾指针(r 指向队尾元素)的值分别为8和3，则该队列中的元素个数为________。

A. 5　　B. 6　　C. 16　　D. 17

答：这里MaxSize＝21，其中的元素个数＝($r-f$＋MaxSize)%MaxSize＝16。本题的答案为C。

29. 设环形队列中数组的下标是0～$N-1$，已知其队头指针 f(f 指向队首元素的前一个位置)和队中元素个数 n，则队尾指针 r(r 指向队尾元素的位置)为________。

A. $f-n$　　B. $(f-n)\%$N

C. $(f+n)\%$N　　D. $(f+n+1)\%$N

答：C。

30. 设环形队列中数组的下标是0～$N-1$，已知其队尾指针 r(r 指向队尾元素的位置)和队中元素个数 n，则队尾指针 f(f 指向队头元素的前一个位置)为________。

A. $r-n$　　B. $(r-n)\%N$

C. $(r-n+N)\%N$　　D. $(r+n)\%N$

答：C。

31. 若用一个大小为6的数组来实现环形队列，rear作为队尾指针指向队列中的尾部元素，front作为队头指针指向队头元素的前一个位置。现在rear和front的值分别是0和3，当从队列中删除一个元素再加入两个元素后rear和front的值分别是________。

A. 1和5　　B. 2和4　　C. 4和2　　D. 5和1

答：删除一个元素时 front 循环增 1，进队两个元素时 rear 循环增 2。本题的答案为 B。

32. 有一个环形队列 qu(存放元素位置 0～MaxSize－1)，rear 作为队尾指针指向队列中的尾部元素，front 作为队头指针指向队头元素的前一个位置，则队满的条件是________。

A. qu－>front＝＝qu－>rear

B. qu－>front＋1＝＝qu－>rear

C. qu－>front＝＝(qu－>rear＋1)％MaxSize

D. qu－>rear＝＝(qu－>front＋1)％MaxSize

答：根据环形队列的结构很快可以排除选项 A 和 B(因为它们与 MaxSize 无关)。环形队列中约定，当进队一个元素后到达了队头就表示队满，进队操作是 rear 循环增 1。本题的答案为 C。

33. 假设用 $Q[0..M]$实现环形队列，f 作为队头指针指向队头元素的前一个位置，r 作为队尾指针指向队尾元素。若用“$(r+1)\%(M+1)==f$”作为队满的标志，则________。

A. 可用“$f==r$”作为队空的标志

B. 可用“$f>r$”作为队空的标志

C. 可用“$(f+1)\%(M+1)==r$”作为队空的标志

D. 队列中最多可以有 $M+1$ 个元素

答：这里 MaxSize 等于 $M+1$，若用“$(r+1)\%(M+1)==f$”作为队满的标志，队列中最多可以有 M 个元素。本题的答案为 A。

34. 环形队列存放在一维数组 $A[0..M-1]$中，end1 指向队头元素，end2 指向队尾元素的后一个位置。假设队列两端均可以进行入队和出队操作，队列中最多能容纳 $M-1$ 个元素，初始时为空。下列判断队空和队满的条件中正确的是________。

A. 队空：end1＝＝end2；队满：end1＝＝(end2＋1) mod M

B. 队空：end1＝＝end2；队满：end2＝＝(end1＋1) mod (M－1)

C. 队空：end2＝＝(end1＋1) mod M；队满：end1＝＝(end2＋1) mod M

D. 队空：end1＝＝(end2＋1) mod M；队满：end2＝＝(end1＋1) mod (M－1)

答：这里环形队列是让队头指针指向队头元素、队尾指针指向队尾元素的后一个位置，和经典做法让队头指针指向队头元素的前一个位置、队尾指针指向队尾元素，在判断队空和队满的条件上是相同的，都是通过少放一个元素来区分队空和队满。本题的答案为 A。

35. 若用 data$[0..n-1]$数组来实现环形队列，初始时队头指针 front(指向队头元素的前一个位置)和队尾指针 rear(指向队列中的尾部元素)均为 0，现有 1～6 的 6 个元素进队，然后出队 8 次，发现原来存放元素 4 的位置变为队头，则 n 为________。

A. 5　　B. 4　　C. 8　　D. 10

答：初始时，front＝rear＝0，进队 1～6 元素，此时元素 4 的位置为 3，出队 8 次后，队头指针 front＝(front＋8)％n＝8％n，即 8％n＝3，则 n＝5。本题的答案为 A。

36. 假设用一个不带头结点的单链表表示队列，队尾应该在链表的________位置。

A. 链头　　B. 链尾　　C. 链中　　D. 以上都可以

答：B。

37. 最适合用作链队的链表是________。

A. 带队首指针和队尾指针的循环单链表

B. 带队首指针和队尾指针的非循环单链表

C. 只带队首指针的非循环单链表

D. 只带队首指针的循环单链表

答：对于链队，进队操作是在队尾插入结点，出队操作是删除队首结点。对于带队首指针和队尾指针的非循环单链表，这两种操作的时间复杂度均为 $O(1)$，所以本题的答案为 B。

38. 最不适合用作链队的链表是________。

A. 只带队首指针的非循环双链表　　B. 只带队首指针的循环双链表

C. 只带队尾指针的循环双链表　　D. 只带队尾指针的循环单链表

答：选项 A 和 B 均可用作链式队列，但不必采用循环单链表，这样反而降低了队列基本运算的效率。本题的答案为 A。

3.3.2 填空题

1. 栈是一种具有________特性的线性表。

答：后进先出或先进后出。

2. 设栈 S 和队列 Q 的初始状态均为空，元素 a,b,c,d,e,f,g 依次进栈 S。若每个元素出栈后立即进入队列 Q，且 7 个元素出列的顺序是 b,d,c,f,e,a,g，则栈 S 的容量至少是________。

答：3。由于队列不改变进出序列，这里变为求通过一个栈将 a,b,c,d,e,f,g 序列变为 b,d,c,f,e,a,g 序列时栈空间至少多大？从其进出栈过程可以看到，栈中最多有 3 个元素，即栈大小至少为 3。

3. 一个初始输入序列 $1,2,\cdots,n$，出栈序列是 $p_1,p_2,\cdots,p_n$，若 $p_1=1$，则 p_2的可能取值个数为________。

答：$n-1$。p_2不可能取值 1，其他 $2\sim n$ 的值都可能。

4. 一个初始输入序列 $1,2,\cdots,n$，出栈序列是 $p_1,p_2,\cdots,p_n$，若 $p_1=4$，则 p_2的可能取值个数为________。

答：$n-3$。p_2不可能取值 4、2、1，其他值都可能。

5. 栈的常用运算是进栈和出栈，设计栈的一种好的存储结构应尽可能保证进栈和出栈运算的时间复杂度为________。

答：$O(1)$。

6. 当利用大小为 n 的数组 data[0..$n-1$]存储一个顺序栈时，假定用 top==n 表示栈空，则向这个栈插入一个元素时首先应执行________语句修改 top 指针。

答：top--。

7. 当利用大小为 n 的数组 data[0..$n-1$]存储一个顺序栈时，假定用 top==-1 表示栈空，则向这个栈插入一个元素时首先应执行________语句修改 top 指针。

答：top++。

8. 若用 data[1..m]作为顺序栈的存储空间，栈空的标志是栈顶指针 top 的值等于 $m+1$，则每进行一次__①__操作，需将 top 的值加 1；每进行一次__②__操作，需将 top 的值减 1。

答：①出栈　②进栈。这里以 data[m]端作为栈底、data[1]端作为栈顶。

9. 当两个栈共享一个存储区时,栈利用一维数组 data[1..n]表示,栈 1 在低下标处,栈 2 在高下标处。两栈顶指针为 top1 和 top2,初始值分别为 0 和 $n+1$,则当栈 1 空时 top1 为__①__,栈 2 空时__②__,栈满时为__③__。

答:①top1=0 ②top2=$n+1$ ③top1+1=top2。

10. 表达式"a+((b * c-d)/e+f * g/h)+i/j"的后缀表达式是________。

答:a b c * d - e / f g * h / + + i j / +。

11. 如果栈的最大长度难以估计,则其存储结构最好使用________。

答:链栈。

12. 若用带头结点的单链表 st 来表示链栈,则栈空的标志是________。

答:st->next==NULL。

13. 若用不带头结点的单链表 st 来表示链式栈,则创建一个空栈所要执行的操作是________。

答:st=NULL。

14. 在用栈求解迷宫路径时,当找到出口时,栈中所有方块________。

答:构成一条迷宫路径。

15. 若用 Q[1..m]作为非环形顺序队列的存储空间,则最多只能执行________次进队操作。

答:m。

16. 若用 Q[1..100]作为环形队列的存储空间,f、r 分别表示队头和队尾指针,f 指向队头元素的前一个位置,r 指向队尾元素,则当 $f=70$,$r=20$ 时,队列中共有________个元素。

答:50。这里 MaxSize=100,元素个数=($r-f$+MaxSize)%MaxSize=50。

17. 环形队列用数组 $A[m..n]$($m<n$)存储元素,其中队头指针 f 指向队头元素的前一个位置、队尾指针 r 指向队尾元素,则该队列中的元素个数是________。

答:$(r-f+n-m+1)\%(n-m+1)$。

18. 用一个大小为 8 的数组来实现环形队列,队头指针 front 指向队头元素的前一个位置,队尾指针 rear 指向队尾元素位置。当前 front 和 rear 的值分别为 0 和 5,现在进队 3 个元素,又出队 3 个元素,front 和 rear 的值分别是________。

答:3,0。这样操作后,front=(0+3)%8=3,rear=(5+3)%8=0。

19. 在实现顺序队的时候,通常将数组看成是一个首尾相连的环,这样做的目的是为了避免产生________现象。

答:假溢出。

20. 已知环形队列的存储空间大小为 m,队头指针 front 指向队头元素、队尾指针 rear 指向队尾元素,则在队列不满的情况下队中元素个数是________。

答:(rear-front+1+m)%m。这样的队列当 front=rear 时,队中有一个元素。

21. 假设用一个不带头结点的单链表表示队列,进队结点 p 的操作是________。

答:将结点 p 插入到单链表末尾。

22. 假设用一个不带头结点的单链表表示队列,非空队列的出队操作是________。

答:删除单链表的首结点。

3.3.3 判断题

1. 判断以下叙述的正确性。

(1) 栈底元素是不能删除的元素。

(2) 顺序栈中元素值的大小是有序的。

(3) 在 n 个元素连续进栈以后,它们的出栈顺序和进栈顺序一定正好相反。

(4) 栈顶元素和栈底元素有可能是同一个元素。

(5) 若用 data[1..m]表示顺序栈的存储空间,则对栈的进栈、出栈操作最多只能进行 m 次。

(6) 栈是一种对进栈、出栈操作总次数做了限制的线性表。

(7) 对顺序栈进行进栈、出栈操作不涉及元素的前、后移动问题。

(8) n 个元素通过一个栈产生 n 个元素的出栈序列,其中进栈和出栈操作的次数总是相等的。

(9) 空的顺序栈没有栈顶指针。

(10) n 个元素进队的顺序和出队的顺序总是一致的。

(11) 环形队列中有多少个元素可以根据队首指针和队尾指针的值来计算。

(12) 若采用"队首指针和队尾指针的值相等"作为环形队列为空的标志,则在设置一个空队时只需将队首指针和队尾指针赋同一个值,不管什么值都可以。

(13) 无论是顺序队还是链队,插入、删除运算的时间复杂度都是 $O(1)$。

(14) 若用不带头结点的非循环单链表来表示链队,则可以用"队首指针和队尾指针的值相等"作为队空的标志。

答:(1) 错误。栈底元素可以出栈即删除。

(2) 错误。顺序栈是指用顺序存储结构实现的栈,栈中的元素不一定是有序的。

(3) 正确。后进栈的元素先出栈,先进栈的元素后出栈,注意这里是连续进栈。

(4) 正确。当栈中只有一个元素时就是这种情况。

(5) 错误。可以进行任意多次交替的进栈、出栈操作,但栈中最多只有 m 个元素。

(6) 错误。栈是只能在同一端进行进栈、出栈操作的线性表。

(7) 正确。

(8) 正确。

(9) 错误。空栈指栈中没有元素,但顺序栈一定要有栈顶指针。

(10) 正确。后进队的元素后出队,先进队的元素先出队。

(11) 正确。

(12) 正确。因为无论出队和入队都要进行求余运算,将队首指针和队尾指针转化为有效的顺序队下标值,所以是正确的。

(13) 正确。

(14) 错误。应该用"队首指针和队尾指针的值均为 NULL"作为队空的标志,因为当链队中只有一个结点时队首指针和队尾指针的值也相等。

2. 判断以下叙述的正确性。

(1) 栈和线性表是两种不同的数据结构,它们的数据元素的逻辑关系也不同。

(2) 有 n 个不同的元素通过一个栈,产生的所有出栈序列恰好构成这 n 个元素的全排列。

(3) 对于 1、2、…、n 的 n 个元素通过一个栈,则以 n 为第一个元素的出栈序列只有一种。

(4) 在顺序栈中,将栈底放在数组的任意位置不会影响运算的时间性能。

(5) 若用 s[1..m]表示顺序栈的存储空间,以 s[1]为栈底,变量 top 指向栈顶元素的前一个位置,当栈未满时,将元素 e 进栈的操作是 top——;s[top]=e。

(6) 在采用单链表作为链栈时必须带有头结点。

(7) 环形队列不存在空间上溢出的问题。

(8) 在队空间大小为 n 的环形队列中最多只能进行 n 次进队操作。

(9) 顺序队采用数组存放队中元素,而数组具有随机存取特性,所以在顺序队中可以随机存取元素。

(10) 对于链队,可以根据队头、队尾指针的值计算队中元素的个数。

答:(1) 错误。

(2) 错误。

(3) 正确。

(4) 错误。在顺序栈中,如果将栈底放在数组的两端,其进栈、出栈运算的时间性能都是最好的。如果将栈底放在数组的中间,要么将数组改为循环的(需要保存该栈底位置),要么移动元素,其时间性能都不如将栈底放在数组两端。

(5) 错误。以 s[1]为栈底,进栈操作时 top 应远离栈底方向移动,所以进栈操作为"s[top]=e ;top++;"。

(6) 错误。可以用不带头结点的单链表表示链栈。

(7) 错误。环形队列只是不存在假溢出,它仍然存在上溢出的问题。

(8) 错误。

(9) 错误。顺序队采用数组存放队中元素,尽管数组具有随机存取特性,但队列的操作特性是顺序存取,只能存取两端的元素。

(10) 错误。

3.3.4 简答题

1. 试各举一个实例,简要说明栈和队列在程序设计中所起的作用。

答:栈的特点是先进后出,所以在解决的实际问题涉及后进先出的情况时可以考虑使用栈。例如求解表达式括号匹配问题时通常使用一个栈,将读到的左括号进栈,每读入一个右括号,判断栈顶是否为左括号,若是,则出栈;否则,表示不匹配。

队列的特点是先进先出。例如求解操作系统中的作业排队问题时通常使用队列,因为在允许多道程序运行的计算机系统中同时有几个作业运行,如果运行的结果都需要通过通道输出,那就要按请求输出的先后次序排队。每当通道传输完毕并可以接受新的输出任务时,队头的作业先从队列中退出做输出操作(出队)。凡是申请输出的作业都从队尾进入队列(进队)。

2. 假设有 4 个元素 a、b、c、d 依次进栈,进栈和出栈操作可以交替进行,试写出所有可

能的出栈序列。

答：当进栈的元素为 n 个时，经过栈运算后可得到的输出序列个数为 $\frac{1}{n+1}C_{2n}^{n}$，其中 $C_{2n}^{n}=\frac{(2n)!}{n!n!}$。这里 $n=4$ 时，出栈序列个数为 $\frac{1}{5}\times\frac{8!}{4!\times4!}=14$ 种，如表 3.1 所示。

表 3.1 出栈序列

以 a 开头	abcd	abdc	acbd	acdb	adcb
以 b 开头	bacd	badc	bcad	bcda	bdca
以 c 开头	cbad	cbda	cdba		
以 d 开头	dcba				

3. 假设以 S 和 X 分别表示进栈和出栈操作，则初态和终态均为栈空的进栈和出栈的操作序列，可以表示为仅由 S 和 X 组成的序列，称可以实现的栈操作序列为合法序列（例如 $SSXX$ 为合法序列，而 $SXXS$ 为非法序列）。试给出区分给定序列为合法序列或非法序列的一般准则，并证明对同一输入序列的两个不同的合法序列不可能得到相同的输出序列。

答：合法的栈操作序列必须满足以下两个条件。

(1) 在操作序列的任何前缀（从开始到任何一个操作时刻）中，S 的个数不得少于 X 的个数。

(2) 整个操作序列中 S 和 X 的个数相等。

要求证明：对同一输入序列 $a_1a_2\cdots a_n$ 的两个不同的合法操作序列 $p=p_1p_2\cdots p_{j-1}p_j\cdots p_{2n}$，$q=q_1q_2\cdots q_{j-1}q_j\cdots q_{2n}$，不可能得到相同的输出序列。

证明：因为 $p\neq q$，所以一定存在一个 $j(1\leqslant j\leqslant 2n)$，使得 $p_1p_2\cdots p_{j-1}=q_1q_2\cdots q_{j-1}$，而 $p_j\neq q_j$，假设操作子序列 $p_1p_2\cdots p_{j-1}$ 已将 $a_1a_2\cdots a_{i-1}$ 进栈且将其中的某些元素出栈，而 $a_ia_{i+1}\cdots a_n$ 尚未进栈。

因为 p 和 q 都是合法的栈操作序列，且 $p_j\neq q_j$，所以 p_j 和 q_j 中必有一个为 S 操作，另一个为 X 操作（不失一般性，不妨设 p_j 为 S 操作，q_j 为 X 操作），而且栈不必为空（否则就不能进行 X 操作）。设栈顶元素为 $a_f(1\leqslant f\leqslant i)$。因此对于操作序列 p 来说，在其对应的输出序列中 a_i 必领先于 a_f（因为 p_j 为 S 操作，它使 a_i 进栈而 a_f 尚在栈中），对于操作序列 q 来说，在其对应的输出序列中 a_f 必领先于 a_i（因为 q_j 为 X 操作，它使 a_f 出栈而 a_i 尚未进栈），所以 p 和 q 必定对应不同的输出序列。

4. 什么是队列的上溢现象和假溢出现象？解决假溢出有哪些方法？

答：在队列的顺序存储结构中，设队头指针为 front、队尾指针为 rear、队的容量（存储空间的大小）为 MaxSize。当有元素进队时，若 rear＝MaxSize（初始时 rear＝0），则发生队列的上溢现象，不能做进队操作。所谓队列假溢出现象是指队列中还有剩余空间但元素却不能进入队列，造成这种现象的原因是队列的设计不合理。

解决队列假溢出的方法有以下几种：

(1) 建立一个足够大的存储空间，但这样做会造成空间的使用效率降低。

(2) 当出现假溢出时可采用以下几种方法。

① 采用平移元素的方法：每当进队一个元素时，队列中已有的元素向队头移动一个位

置(当然要有空闲的空间可供移动)。这种方法对应进队运算的时间复杂度为 $O(n)$。

② 每当出队一个元素时,依次移动队中的元素,始终使 front 指针指向队列中的第一个元素位置。这种方法对应出队运算的时间复杂度为 $O(n)$。

③ 采用环形队列方式:把队列看成一个首尾相接的环形队列,在环形队列上进行进队或出队运算时仍然遵循"先进先出"的原则。这种方法对应进队和出队运算的时间复杂度均为 $O(1)$。

5. 在利用两个栈 S_1、S_2模拟一个队列时如何用栈的基本运算实现队列的进队、出队以及队列的判空等基本运算,请简述算法思想。

答:利用两个栈 S_1和 S_2模拟一个队列的基本思想是用栈 S_1作为输入栈、栈 S_2作为输出栈。进队时,总是将元素进栈到 S_1,出队时,若输出栈 S_2已空,则将 S_1中的元素全部出栈到 S_2中,然后由 S_2出栈元素。若输出栈 S_2不空,则直接由 S_2出栈元素。显然,只有当输入栈、输出栈均为空时队列才为空。

6. 设输入元素为 1、2、3、P 和 A,输入次序为 123PA,元素经过一个栈后产生输出序列,在所有输出序列中有哪些序列可作为高级语言的变量名(以字母开头的字母数字串)。

答:AP321,PA321,P3A21,P32A1,P321A。

7. 用栈实现将中缀表达式 8-(3+5)*(5-6/2)转换成后缀表达式,画出栈的变化过程图。

答:栈的变化过程如表 3.2 所示。最后生成的后缀表达式为 8 3 5 + 5 6 2 / - * -,其求值结果为-8。

表 3.2 将中缀表达式 8-(3+5)*(5-6/2)转换成后缀表达式时栈的变化过程

op 栈	postexp	说明
	8#	将 8# 存入 postexp 中
-	8#	'-'进栈
-(	8#	'('进栈
-(	8#3#	将 3# 存入 postexp 中
-(+	8#3#	'+'进栈
-(+	8#3#5#	将 5# 存入 postexp 中
-	8#3#5#+	遇到')',将'+'和'('出栈
-*	8#3#5#+	'*'进栈
-*(	8#3#5#+	'('进栈
-*(	8#3#5#+5#	将 5# 存入 postexp 中
-*(-	8#3#5#+5#	'-'进栈
-*(-	8#3#5#+5#6#	将 6# 存入 postexp 中
-*(-/	8#3#5#+5#6#	'/'进栈
-*(-/	8#3#5#+5#6#2#	将 2# 存入 postexp 中
-*	8#3#5#+5#6#2#/-	遇到')',将'/''-'和'('出栈
	8#3#5#+5#6#2#/-*-	exp 扫描完毕,将栈中的所有运算符依次出栈并存入数组 postexp 中,得到后缀表达式

8. 简述以下算法的功能：

```
void fun(int a[ ],int n)
{   int i = 0,e;
    SqStack * st;
    InitStack(st);
    for (i = 0;i < n;i++)
        Push(st,a[i]);
    i = 0;
    while (!StackEmpty(st))
    {   Pop(st,e);
        a[i++] = e;
    }
    DestroyStack(st);
}
```

答：算法的执行步骤如下。

(1) 扫描整数数组 a，将所有元素进到 st 栈中。

(2) 将 st 的所有元素退栈，放到数组 a 中并覆盖原有位置的元素，从而将数组 a 的所有元素逆置。

(3) 销毁栈 st。

所以本算法的功能是利用栈 st 将数组 a 中的所有元素逆置。

9. 阅读以下程序，给出其输出结果：

```
char * fun(int d)
{   char e;int i = 0,x;
    static char b[MaxSize];
    SqStack * st;
    InitStack(st);
    while (d!= 0)
    {   x = d % 16;
        if (x < 10)e = '0' + x;
        elsee = 'A' + x - 10;
        Push(st,e);
        d/ = 16;
    }
    while (!StackEmpty(st))
    {   Pop(st,e);
        b[i++] = e;
    }
    b[i] = '\0';
    DestroyStack(st);
    return b;
}
int main()
{   int d = 1000,i;
    char * b;
    b = fun(d);
```

```
        for (i=0;b[i];i++)
            printf("%c",b[i]);
        printf("\n");
        return 1;
    }
```

答：fun(d)算法的功能是采用辗转相除法将十进制数 d 转换成十六进制数，并用数组 b 存放十六进制数串。本程序的功能是将十进制数 1000 转换成十六进制数并输出，其结果为 3E8。

10. 算法 fun 的功能是借助栈结构实现整数从十进制到八进制的转换，阅读算法并回答问题：

(1) 画出 n 为十进制数 1348 时算法执行过程中栈的动态变化情况。

(2) 说明算法中 while 循环完成的操作。

```
void fun(int n)                    //n为非负的十进制整数
{   int e;
    SqStack *S;
    InitStack(S);
    do
    {   Push(S,n%8);
        n=n/8;
    } while (n);
    while (!StackEmpty(S))
    {   Pop(S,e);
        printf("%ld",e);
    }
}
```

答：(1) n 为十进制数 1348 时算法执行过程中栈的动态变化情况如图 3.4 所示，产生对应的八进制数为 2504。

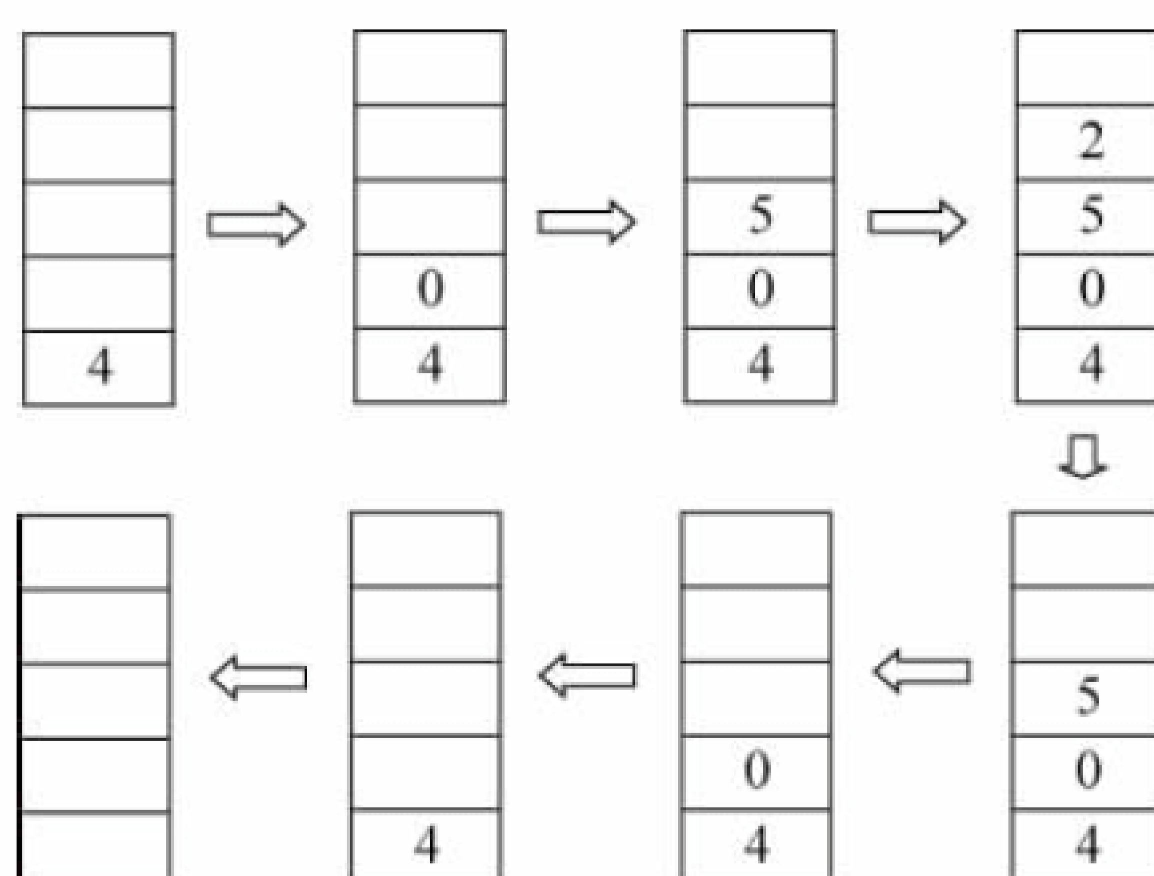


图 3.4　栈的动态变化情况

(2) 算法中 while 循环的操作是退栈所有元素并输出，即从高位到低位输出八进制数。

11. 简述以下算法的功能(栈的元素类型为 int)。

```
void fun(SqStack *&st)
{   int i,j=0,A[MaxSize];
```

```
        while (!StackEmpty(st))
        {   Pop(S,A[j]);
            j++;
        }
        for(i = 0;i < j;i++)
            Push(S,A[i]);
    }
```

答：算法的执行步骤如下。

(1) 将栈 st 中的所有元素退栈并存放到数组 A 中。

(2) 将 A 中的元素一一进栈，达到逆置栈 st 的目的。

所以本算法的功能是逆置栈 st 的所有元素。

3.3.5 算法设计题

1.【顺序栈算法】设计一个算法将一个十进制正整数 d 转换为相应的二进制数。

解：将十进制正整数转换成二进制数通常采用除 2 取余数法。在转换过程中，二进制数是按照从低位到高位的次序得到的，这和通常的从高位到低位输出二进制的次序相反。为此设计一个栈 st，用于暂时存放每次得到的余数，当转换过程结束时，退栈所有元素便得到从高位到低位的二进制数。图 3.5 所示为十进制数 12 转换为二进制数 1100 的过程。

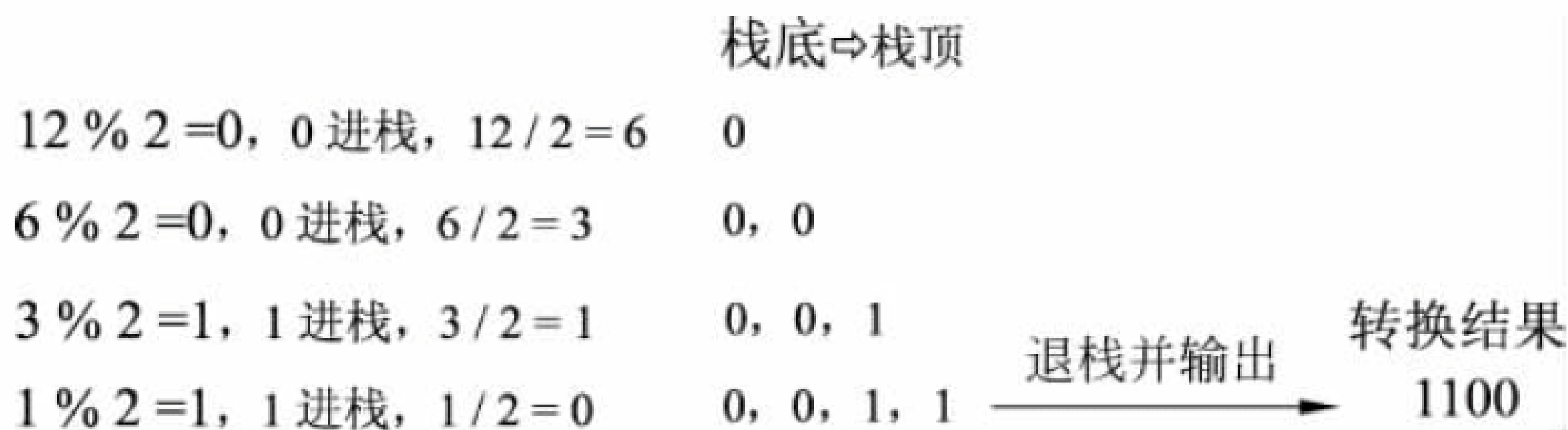


图 3.5 整数 12 转换为二进制数的过程

对应的算法如下：

```
#include "SqStack.cpp"              //包含顺序栈的定义及运算函数
void trans(int d,char b[])          //b 用于存放 d 转换成的二进制数串
{   char e;
    SqStack *st;
    InitStack(st);
    int i = 0;
    while (d!= 0)
    {   e = '0' + d % 2;            //求余数并转换为字符
        Push(st,e);
        d/ = 2;                     //继续求更高位
    }
    while (!StackEmpty(st))
    {   Pop(st,e);                  //出栈元素 e
        b[i] = e;                   //将 e 存放在数组 b 中
        i++;
    }
```

```
    b[i] = '\0';                    //加入字符串结束标志
    DestroyStack(st);               //销毁栈
}
```

2.【顺序栈算法】设计一个算法，利用顺序栈的基本运算从栈顶到栈底输出栈中的所有元素，要求仍保持栈中元素不变。

解：先建立并初始化一个临时栈 tmpst。退栈 st 中的所有元素，输出这些元素并进栈到 tmpst 中，然后将临时栈 tmpst 中的元素逐一出栈并进栈到 st 中，这样恢复 st 栈中原来的元素。注意本题要求只能使用栈的基本运算来完成，不能直接用 st—>data[i]输出栈中的元素。对应的算法如下：

```
#include "SqStack.cpp"                  //包含顺序栈的定义及运算函数
void DispStack(SqStack *st)
{   ElemType x;
    SqStack *tmpst;                     //定义临时栈
    InitStack(tmpst);                   //初始化临时栈
    while (!StackEmpty(st))             //临时栈 tmpst 中包含 st 栈中的逆转元素
    {   Pop(st,x);
        printf("%d ",x);
        Push(tmpst,x);
    }
    printf("\n");
    while (!StackEmpty(tmpst))          //恢复 st 栈中原来的内容
    {   Pop(tmpst,x);
        Push(st,x);
    }
    DestroyStack(tmpst);
}
```

3.【顺序栈算法】设计一个算法，利用顺序栈的基本运算求栈中从栈顶到栈底的第 k 个元素，要求仍保持栈中元素不变。

解：先建立并初始化一个临时栈 tmpst。退栈 st 中的所有元素 x，并用 i 累计元素个数，当 $i==k$ 时置 $e=x$，并将所有元素进栈到 tmpst 中，然后将临时栈 tmpst 中的元素逐一出栈并进栈到 st 中，这样恢复 st 栈中原来的元素。如果栈中没有第 k 个元素，返回假；否则返回真，并通过引用型参数 e 保存第 k 个元素。注意本题要求只能使用栈的基本运算来完成，不能直接用 st—>data[i]求第 k 个栈中元素。对应的算法如下：

```
#include "SqStack.cpp"              //包含顺序栈的定义及运算函数
bool Findk(SqStack *st, int k, ElemType &e)
{   int i = 0;
    bool flag = false;
    ElemType x;
    SqStack *tmpst;                 //定义临时栈
    InitStack(tmpst);               //初始化临时栈
    while (!StackEmpty(st))         //临时栈 tmpst 中包含 st 栈中的逆转元素
    {   i++;
        Pop(st,x);
```

```
        if (i==k)
        {   e=x;
            flag=true;
        }
        Push(tmpst,x);
    }
    while (!StackEmpty(tmpst))      //恢复 st 栈中原来的内容
    {   Pop(tmpst,x);
        Push(st,x);
    }
    DestroyStack(tmpst);
    return flag;
}
```

4.【顺序栈算法】有 abcde 共 $n(n=5)$ 个字符，通过一个栈可以产生多种出栈序列，设计一个算法判断序列 str 是否为一个合适的出栈序列，并给出操作过程，要求用相关数据进行测试。

解：先建立一个字符顺序栈 st，将进栈序列 abcde 存放到字符数组 A 中。用 i、j 分别扫描数组 A 和 str，它们的初始值均为 0。当数组 A 和 str 都没有扫描完时循环：比较栈顶元素 e 和 str[j]，若两者不相同，则将 $A[i]$ 进栈，i 加 1；若两者相同，则出栈栈顶元素 e，j 加 1。上述循环结束后退栈所有元素。如果序列 str 是一个合适的出栈序列，必有 $j==n$，否则 str 不是一个合适的出栈序列。对应的算法如下：

```
#include "SqStack.cpp"              //包含顺序栈的定义及运算函数
bool isSerial(char str[],int n)
{   int i,j;
    char A[MaxSize],e;
    SqStack *st;                    //建立一个顺序栈
    InitStack(st);
    for (i=0;i<n;i++)
        A[i]='a'+i;                 //将 abcde 放入数组 A 中
    i=0;j=0;
    while (i<n && j<n)
    {   if (StackEmpty(st) || (GetTop(st,e) && e!=str[j]))
        {   Push(st,A[i]);
            printf("  元素%c进栈\n",A[i]);
            i++;
        }
        else
        {   Pop(st,e);
            printf("  元素%c出栈\n",e);
            j++;
        }
    }
    while (!StackEmpty(st) && GetTop(st,e) && e==str[j])
    {   Pop(st,e);
        printf("  元素%c出栈\n",e);
```

```
            j++;
        }
        DestroyStack(st);
        if (j==n) return true;          //是出栈序列时返回 true
        else return false;              //不是出栈序列时返回 false
}
void Disp(char str[],int n)             //输出 str
{   int i;
    for (i=0;i<n;i++)
        printf("%c",str[i]);
}
int main()
{   int n=5;
    char str[]="acbed";
    Disp(str,n); printf("的操作序列:\n");
    if (isSerial(str,n))
    {   Disp(str,n);
        printf("是合适的出栈序列\n");
    }
    else
    {   Disp(str,n);
        printf("不是合适的出栈序列\n");
    }
    return 1;
}
```

本程序的执行结果如下：

```
acbed 的操作系列:
元素 a 进栈
元素 a 出栈
元素 b 进栈
元素 c 进栈
元素 c 出栈
元素 b 出栈
元素 d 进栈
元素 e 进栈
元素 e 出栈
元素 d 出栈
acbed 是合适的出栈序列
```

5.【共享栈算法】用一个一维数组 S(设大小为 MaxSize)作为两个栈的共享空间，说明共享方法，以及栈满、栈空的判断条件，并用 C/C++ 语言设计公用的初始化栈运算 InitStack1(st)、判栈空运算 StackEmpty1(st,i)、进栈运算 Push1(st,i,x)和出栈运算 Pop1(st,i,x)，其中 i 为 1 或 2，用于表示栈号，x 为进栈或出栈元素。

解：设用一维数组 S[MaxSize]作为两个栈 $S1$ 和 $S2$ 的共享空间，整型变量 top1、top2 分别作为两个栈的栈顶指针，并约定栈顶指针指向当前元素的下一个位置。$S1$ 的栈底位置设在 S[0]，$S2$ 的栈底位置设在 S[MaxSize−1]，如图 3.6 所示。



图 3.6　共享栈示意图

栈 S1 空的条件是 top1==-1，栈 S1 满的条件是 top1==top2-1；栈 S2 空的条件是 top2==MaxSize，栈 S2 满的条件是 top2==top1+1。归纳起来，栈 S1 和 S2 满的条件都是 top1==top2-1。

元素 x 进栈 S1 的算法是 Push1(&st,1,x)，当不满时，执行 st.top1++，st.S[st.top1]=x；元素 x 进栈 S2 的算法是 Push1(&st,2,x)，当不满时，执行 st.top2--，st.S[st.top2]=x。

元素 x 退栈 S1 的算法是 Pop1(&st,1,&x)，当不空时，执行 x=st.S[st.top1]，st.top1--；元素 x 退栈 S2 的算法是 Pop1(&st,2,&x)，当不空时，执行 x=st.S[st.top2]，st.top2++。

共享栈的类型定义和相关运算算法如下：

```
#include <stdio.h>
#define MaxSize 100
typedef char ElemType;
typedef struct
{   ElemType S[MaxSize];            //存放共享栈中的元素
    int top1,top2;                  //两个栈顶指针
} StackType;                        //声明共享栈类型
//----- 栈初始化算法 ------
void InitStack1(StackType &st)
{   st.top1 = -1;
    st.top2 = MaxSize;
}
//----- 判栈空算法: i=1:栈 1,i=2:栈 2 ------
bool StackEmpty1(StackType st, int i)
{   if (i==1)
        return(st.top1 == -1);
    else                            //i=2
        return(st.top2 == MaxSize);
}
//----- 进栈算法: //i=1:栈 1,i=2:栈 2 ------
bool Push1(StackType &st, int i, ElemType x)
{   if (st.top1 == st.top2 - 1)     //栈满
        return false;
    if (i==1)                       //x 进栈 S1
    {   st.top1++;
        st.S[st.top1] = x;
    }
    else if (i==2)                  //x 进栈 S2
    {   st.top2--;
```

```
            st.S[st.top2] = x;
        }
        else                            //参数 i 错误返回 false
            return false;
        return true;                    //操作成功返回 true
    }
    //----- 出栈算法: i = 1:栈 1,i = 2:栈 2------
    bool Pop1(StackType &st, int i,ElemType &x)
    {   if (i == 1)                     //S1 出栈
        {   if (st.top1 == -1)          //S1 栈空
                return false;
            else                        //出栈 S1 的元素
            {   x = st.S[st.top1];
                st.top1 -- ;
            }
        }
        else if (i == 2)                //S2 出栈
        {   if (st.top2 == MaxSize)     //S2 栈空
                return false;
            else                        //出栈 S2 的元素
            {   x = st.S[st.top2];
                st.top2++;
            }
        }
        else                            //参数 i 错误返回 false
            return false;
        return true;                    //操作成功返回 true
    }
```

6.【环形队列算法】设计一个算法,利用环形队列的基本运算返回指定队列中的队尾元素,要求算法的空间复杂度为 $O(1)$。

解:由于算法要求空间复杂度为 $O(1)$,所以不能使用临时队列。先求出队列 qu 中的元素个数 m。循环 m 次,出队一个元素 x,再将元素 x 进队,最后的 x 即为队尾元素。对应的算法如下:

```
    #include "SqQueue.cpp"          //包含顺序队的类型定义和运算函数
    ElemType Last(SqQueue *qu)
    {   ElemType x;
        int i,m = (qu->rear - qu->front + MaxSize) % MaxSize;
        for (i = 1;i <= m;i++)
        {   deQueue(qu,x);          //出队元素 x
            enQueue(qu,x);          //将元素 x 进队
        }
        return x;
    }
```

7.【环形队列算法】对于环形队列,利用队列的基本运算设计删除队列中从队头开始的第 k 个元素的算法。

解：先求出队列 qu 中的元素个数 count，若 k 小于 0 或大于 count，返回假。出队所有元素，并记录元素的序号 i，当 $i=k$ 时对应的元素只出不进，否则将出队的元素又进队。对应的算法如下：

```
#include "SqQueue.cpp"            //包含顺序队的类型定义和运算函数
bool Delk(SqQueue *&qu,int k)
{   ElemType e;
    int i,count = (qu->rear - qu->front + MaxSize) % MaxSize;
    if (k<=0 || k>count)
        return false;
    for (i=1;i<=count;i++)
    {   deQueue(qu,e);            //出队元素 e
        if (i!=k)                 //第 k 个元素只出不进
            enQueue(qu,e);        //其他元素出队后又进队
    }
    return true;
}
```

说明：在设计本题算法时不能通过移动元素的方式直接对数组 data 删除第 k 个元素，这样是把顺序队看成一个顺序表，没有作为一个队列看待。

8.【环形队列算法】对于环形队列来说，如果知道队尾元素的位置和队列中元素的个数，则队头元素所在的位置显然是可以计算的。也就是说，可以用队列中元素的个数代替队头指针。编写出这种环形顺序队列的初始化、进队、出队和判空算法。

解：当已知队头元素的位置 rear 和队列中元素的个数 count 后，队空的条件为 count==0；队满的条件为 count==MaxSize；计算队头位置为 front=(rear−count+MaxSize)%MaxSize。对应的算法如下：

```
typedef struct
{   ElemType data[MaxSize];
    int rear;                       //队尾指针
    int count;                      //队列中元素的个数
}QuType;                            //队列类型
void InitQu(QuType *&q)             //队列的初始化运算
{   q = (QuType *)malloc(sizeof(QuType));
    q->rear = 0;
    q->count = 0;
}
bool EnQu(QuType *&q,ElemType x)    //进队运算
{   if (q->count == MaxSize)        //队满上溢出
        return false;
    else
    {   q->rear = (q->rear + 1) % MaxSize;
        q->data[q->rear] = x;
        q->count++;
        return true;
    }
}
bool DeQu(QuType *&q,ElemType &x)   //出队运算
```

```
{   int front;                          //局部变量
    if (q->count==0)                    //队空下溢出
        return false;
    else
    {   front=(q->rear-q->count+MaxSize)%MaxSize;
        front=(front+1)%MaxSize;        //队头位置进1
        x=q->data[front];
        q->count--;
        return true;
    }
}
bool QuEmpty(QuType *q)                 //判空运算
{
    return(q->count==0);
}
```

9.【环形队列算法】设计一个环形队列，用 front 和 rear 分别作为队头和队尾指针，另外用一个标志 tag 标识队列可能空(0)或可能满(1)，这样加上 front==rear 可以作为队空或队满的条件，要求设计队列的相关基本运算算法。

解：设计的队列类型如下：

```
typedef struct
{   ElemType data[MaxSize];
    int front,rear;             //队头和队尾指针
    int tag;                    //为0表示队可能空,为1时表示队可能满
} QueueType;
```

初始时 tag=0，front=rear=0，成功的进队操作后 tag=1(任何进队操作后队列可能满，但不一定满，任何进队操作后队列不可能空)，成功的出队操作后 tag=0(任何出队操作后队列可能空，但不一定空，任何出队操作后队列不可能满)，因此这样的队列的 4 要素如下。

① 队空条件：qu.front==qu.rear && qu.tag==0；

② 队满条件：qu.front==qu.rear && qu.tag==1；

③ 元素 x 进队：qu.rear=(qu.rear+1)%MaxSize；qu.data[qu.rear]=x；qu.tag=1；

④ 元素 x 出队：qu.front=(qu.front+1)%MaxSize；x=qu.data[qu.front]；qu.tag=0。

对应的算法如下：

```
void InitQueue1(QueueType &qu)              //初始化队列算法
{   qu.front=qu.rear=0;
    qu.tag=0;                               //为0表示队空可能为空
}
bool QueueEmpty1(QueueType qu)              //判队空算法
{
    return(qu.front==qu.rear && qu.tag==0);
}
bool QueueFull1(QueueType qu)               //判队满算法
```

```
{
    return(qu.tag == 1 && qu.front == qu.rear);
}
bool EnQueue1(QueueType &qu,ElemType x)          //进队算法
{   if (QueueFull1(qu) == 1)                     //队满
        return false;
    qu.rear = (qu.rear + 1) % MaxSize;
    qu.data[qu.rear] = x;
    qu.tag = 1;                                  //至少有一个元素,可能满
    return true;
}
bool DeQueue1(QueueType &qu,ElemType &x)         //出队算法
{   if (QueueEmpty1(qu) == 1)                    //队空
        return false;
    qu.front = (qu.front + 1) % MaxSize;
    x = qu.data[qu.front];
    qu.tag = 0;                                  //出队一个元素,可能空
    return true;
}
```

10.【双端队列应用】假设有一个整型数组存放 n 个学生的分数,将分数分为 3 个等级,分数高于或等于 90 的为 A 等,分数低于 60 的为 C 等,其他为 B 等。要求采用双端队列,先输出 A 等分数,再输出 C 等分数,最后输出 B 等分数。

解:设计双端队列的从队头出队算法 deQueue1、从队头进队算法 enQueue1 和从队尾进队算法 enQueue2。对于含有 n 个分数的数组 a,扫描所有元素 $a[i]$,若 $a[i]$为 A 等,直接输出;若为 B 等,将其从队尾进队;若为 C 等,将其从队头进队。最后从队头出队并输出所有的元素。对应的算法如下:

```
#include "SqQueue.cpp"                               //包含顺序队的类型定义和运算函数
bool deQueue1(SqQueue *&q,ElemType &e)               //从队头出队算法
{   if (q->front == q->rear)                         //队空下溢出
        return false;
    q->front = (q->front + 1) % MaxSize;             //修改队头指针
    e = q->data[q->front];
    return true;
}
bool enQueue1(SqQueue *&q,ElemType e)                //从队头进队算法
{   if ((q->rear + 1) % MaxSize == q->front)         //队满
        return false;
    q->data[q->front] = e;                           //e 元素进队
    q->front = (q->front - 1 + MaxSize) % MaxSize;   //修改队头指针
    return true;
}
bool enQueue2(SqQueue *&q,ElemType e)                //从队尾进队算法
{   if ((q->rear + 1) % MaxSize == q->front)         //队满上溢出
        return false;
    q->rear = (q->rear + 1) % MaxSize;               //修改队尾指针
    q->data[q->rear] = e;                            //e 元素进队
```

```
        return true;
    }
    void fun(int a[ ],int n)
    {   int i;
        ElemType e;
        SqQueue * qu;
        InitQueue(qu);
        for (i=0;i<n;i++)
        {   if (a[i]>=90)
                printf("%d ",a[i]);
            else if (a[i]>=60)
                enQueue2(qu,a[i]);              //从队尾进队
            else
                enQueue1(qu,a[i]);              //从队头进队
        }
        while (!QueueEmpty(qu))
        {   deQueue1(qu,e);                     //从队头出队
            printf("%d ",e);
        }
        printf("\n");
        DestroyQueue(qu);
    }
```

11.【顺序栈和顺序队算法】用于列车编组的铁路转轨网络是一种栈结构，如图 3.7 所示，其中右边轨道是输入端、左边轨道是输出端。当右边轨道上的车皮编号顺序为 1、2、3、4 时，如果执行操作进栈、进栈、出栈、进栈、进栈、出栈、出栈、出栈，则在左边轨道上的车皮编号顺序为 2、4、3、1。

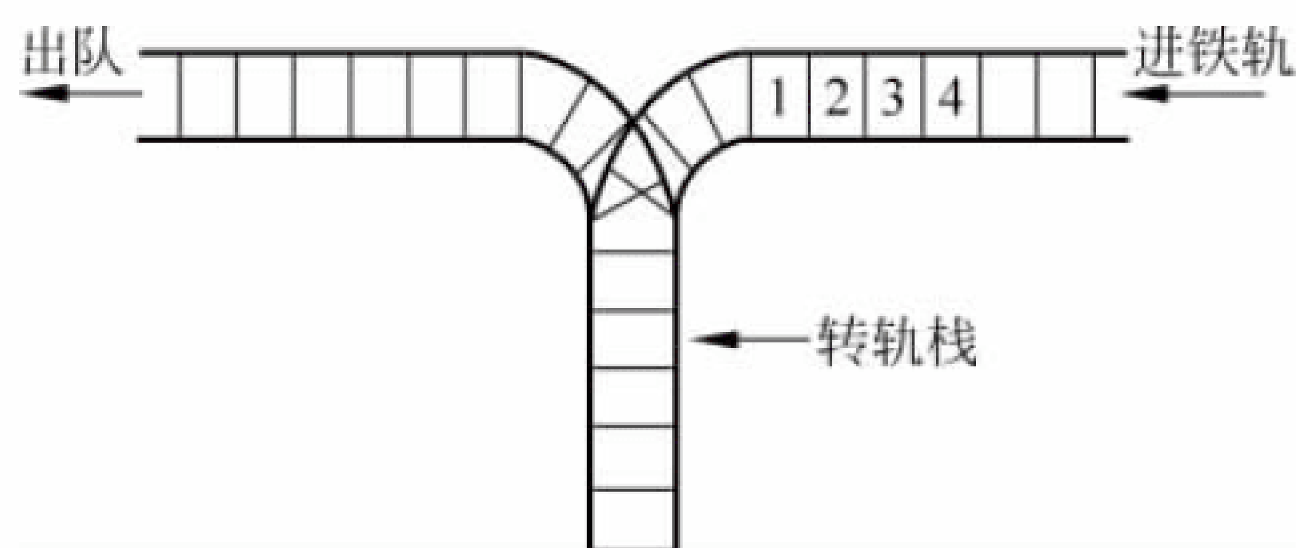


图 3.7　铁路转轨网络

设计一个算法，输入 n 个整数，表示右边轨道上 n 节车皮的编号，用上述转轨栈对这些车皮重新编排，使得编号为奇数的车皮都排在编号为偶数的车皮的前面。

解：将转轨栈看成一个栈，将左边轨道看成是一个队列。从键盘逐个输入表示右边轨道上车皮编号的整数，根据其奇偶性做以下处理：若是奇数，则将其插入到表示左边轨道的顺序队列的队尾；若是偶数，则将其插入到表示转轨栈的顺序栈的栈顶。当 n 个整数都检测完之后，这些整数已全部进入队列或栈中。此时，首先按先进先出的顺序输出队列中的元素，然后再按后进先出的顺序输出栈中的元素。

算法中直接使用两个数组 st 和 qu 分别存放栈和队列中的元素。对应的算法如下：

```
#include <stdio.h>
#define MaxSize 100
void fun1()
{   int i,n,x;
    int st[MaxSize],top = -1;           //顺序栈和栈顶指针
    int qu[MaxSize],front = 0,rear = 0; //队列和队指针
    printf("n:");
    scanf("%d",&n);
    for (i = 0;i<n;i++)
    {   printf("第%d个车皮编号:",i+1);
        scanf("%d",&x);
        if (x%2 == 1)                   //编号为奇数,则进队列
        {   qu[rear] = x;
            rear++;
            printf("  %d进队\n",x);
        }
        else                            //编号为偶数,则进栈
        {   top++;
            st[top] = x;
            printf("  %d进栈\n",x);
        }
    }
    printf("出轨操作:\n  ");
    while (front!= rear)                //队列中的所有元素出队
    {   printf("%d出队 ",qu[front]);
        front++;
    }
    while (top>=0)                      //栈中的所有元素出栈
    {   printf("%d出栈 ",st[top]);
        top--;
    }
    printf("\n");
}
int main()
{   fun1();
    return 1;
}
```

本程序的一次求解结果如下：

```
n:4↙
第 1 个车皮编号:4↙  4 进栈
第 2 个车皮编号:1↙  1 进队
第 3 个车皮编号:3↙  3 进队
第 4 个车皮编号:2↙  2 进栈
出轨操作:
    1 出队  3 出队  2 出栈  4 出栈
```

第4章 串



4.1 本章知识体系



1. 知识结构图

本章的知识结构如图 4.1 所示。

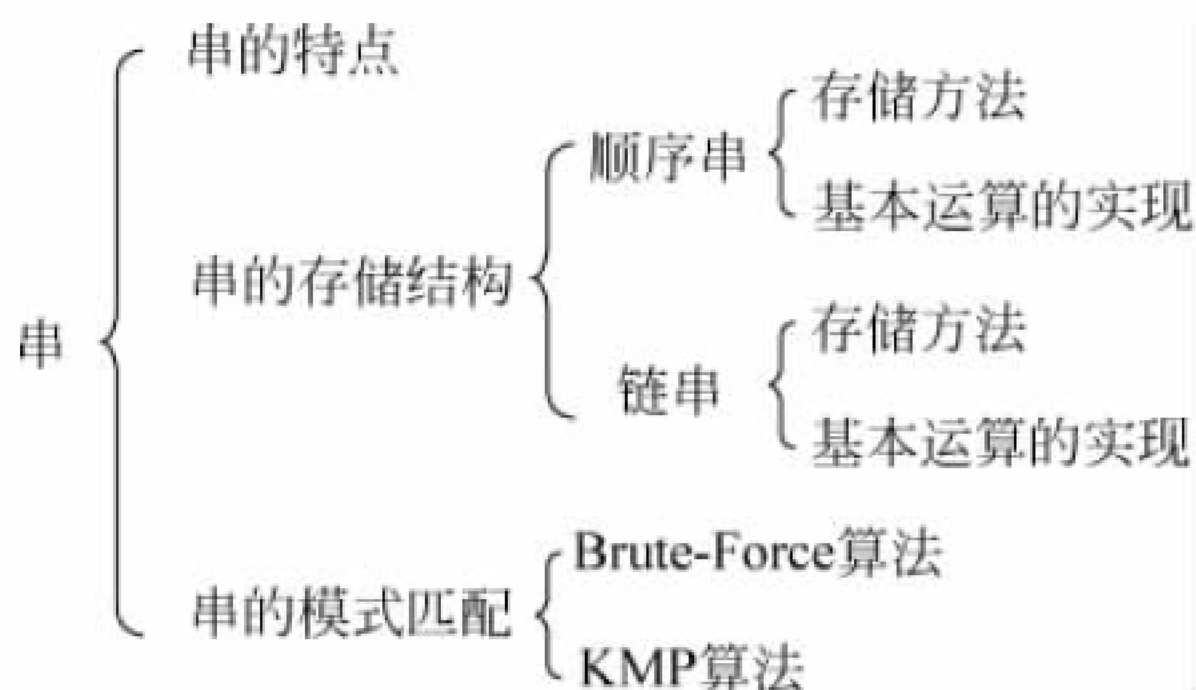


图 4.1　第 4 章知识结构图

2. 基本知识点

(1) 串的相关概念。

(2) 串的顺序存储结构和链式存储结构的优缺点。

(3) 顺序串运算算法设计。

(4) 链串运算算法设计。

(5) BF 模式匹配算法设计。

(6) KMP 算法设计，KMP 算法是提高串匹配效率的。

3. 要点归纳

(1) 串是若干个字符的有限序列，空串是长度为零的串。

(2) 串可以看成是一种特殊的线性表，其逻辑关系为线性关系。

(3) 串的长度是指串中所含字符的个数。

(4) 含 n 个不同字符的串的子串个数为 $n(n+1)/2+1$。

(5) 串主要有顺序串和链串两种存储结构。

(6) 顺序串的算法设计和顺序表类似，链串的算法设计和单链表类似。

(7) 在串匹配中一般将主串称为目标串，将子串称为模式串。

(8) BF 模式匹配算法中需要回溯，时间复杂度为 $O(m\times n)$，而 KMP 算法消除了回溯，时间复杂度为 $O(m+n)$。

4.2 教材中的练习题及参考答案



1. 串是一种特殊的线性表，请从存储和运算两方面分析它的特殊之处。

答：从存储方面看，串中每个元素是单个字符，在设计串存储结构时可以每个存储单元

或者结点只存储一个字符。从运算方面看，串有连接、判断串相等、求子串和子串替换等基本运算，这是线性表的基本运算中所没有的。

2. 为什么在模式匹配中BF算法是有回溯算法，而KMP算法是无回溯算法？

答：设目标串为 s，模式串为 t。在BF算法的匹配过程中，当 $t[j]=s[i]$ 时，置 i++，j++；当 $t[j]\neq s[i]$ 时，置 $i=i-j+1$，$j=0$。从中看到，一旦两个字符不等，目标串指针 i 会回退，所以BF算法是有回溯算法。在KMP算法的匹配过程中，当 $t[j]=s[i]$ 时，置 i++，j++；当 $t[j]\neq s[i]$ 时，i 不变，置 $j=\text{next}[j]$。从中看到，目标串指针 i 不会回退，只会保持位置不变或者向后推进，所以KMP算法是无回溯算法。

3. 在KMP算法中计算模式串的next时，若 $j=0$，为什么要置 $\text{next}[0]=-1$？

答：当模式串中的 t_0 字符与目标串中的某字符 s_i 比较不相等时，置 $\text{next}[0]=-1$ 表示模式串中已没有字符可与目标串中的 s_i 比较，目标串的当前指针 i 应后移至下一个字符，再和模式串中的 t_0 字符进行比较。

4. KMP算法是简单模式匹配算法的改进，以目标串 $s=$"aabaaabc"、模式串 $t=$"aaabc"为例说明的next的作用。

答：模式串 $t=$"aaabc"的next数组值如表4.1所示。

表4.1 模式串 t 对应的next数组

j	0	1	2	3	4
$t[j]$	a	a	a	b	c
$\text{next}[j]$	-1	0	1	2	0

从 $i=0$，$j=0$ 开始，当两者对应字符相等时，i++，j++，直到 $i=2$，$j=2$ 时对应字符不相等。如果是简单模式匹配，下次从 $i=1$，$j=0$ 开始比较。

KMP算法已经获得了前面字符比较的部分匹配信息，即 $s[0..1]=t[0..1]$，所以 $s[0]=t[0]$，而 $\text{next}[2]=1$ 表明 $t[0]=t[1]$，所以有 $s[0]=t[1]$，这说明下次不必从 $i=1$，$j=0$ 开始比较，而只需保持 $i=2$ 不变，让 $i=2$ 和 $j=\text{next}[j]=1$ 的字符进行比较。

$i=2$，$j=1$ 的字符比较不相等，保持 $i=2$ 不变，取 $j=\text{next}[j]=0$。

$i=2$，$j=0$ 的字符比较不相等，保持 $i=2$ 不变，取 $j=\text{next}[j]=-1$。

当 $j=-1$ 时 i++、j++，则 $i=3$，$j=0$，对应的字符均相等，一直比较到 j 超界，此时表示匹配成功，返回3。

从中看到，$\text{next}[j]$ 保存了部分匹配的信息，用于提高匹配效率。由于是在模式串的 j 位置匹配失败的，next也称为失效函数或失配函数。

5. 给出以下模式串的next值和nextval值：

(1) ababaa

(2) abaabaab

答：(1) 求其next和nextval值如表4.2所示。

(2) 求其next和nextval值如表4.3所示。

6. 设目标串 $s=$"abcaabbabcabaacbacba"，模式串 $t=$"abcabaa"。

(1) 计算模式串 t 的nextval数组。

(2) 不写算法，给出利用改进的KMP算法进行模式匹配的过程。

表 4.2 模式串"ababaa"对应的 next 数组

j	0	1	2	3	4	5
$t[j]$	a	b	a	b	a	a
next[j]	−1	0	0	1	2	3
nextval[j]	−1	0	−1	0	−1	3

表 4.3 模式串"abaabaab"对应的 next 数组

j	0	1	2	3	4	5	6	7
$t[j]$	a	b	a	a	b	a	a	b
next[j]	−1	0	0	1	1	2	3	4
nextval[j]	−1	0	−1	1	0	−1	1	0

(3) 总共进行了多少次字符比较?

解:(1) 先计算 next 数组,在此基础上求 nextval 数组,如表 4.4 所示。

表 4.4 计算 next 数组和 nextval 数组

j	0	1	2	3	4	5	6
$t[j]$	a	b	c	a	b	a	a
next[j]	−1	0	0	0	1	2	1
nextval[j]	−1	0	0	−1	0	2	1

(2) 改进的 KMP 算法进行模式匹配的过程如图 4.2 所示。

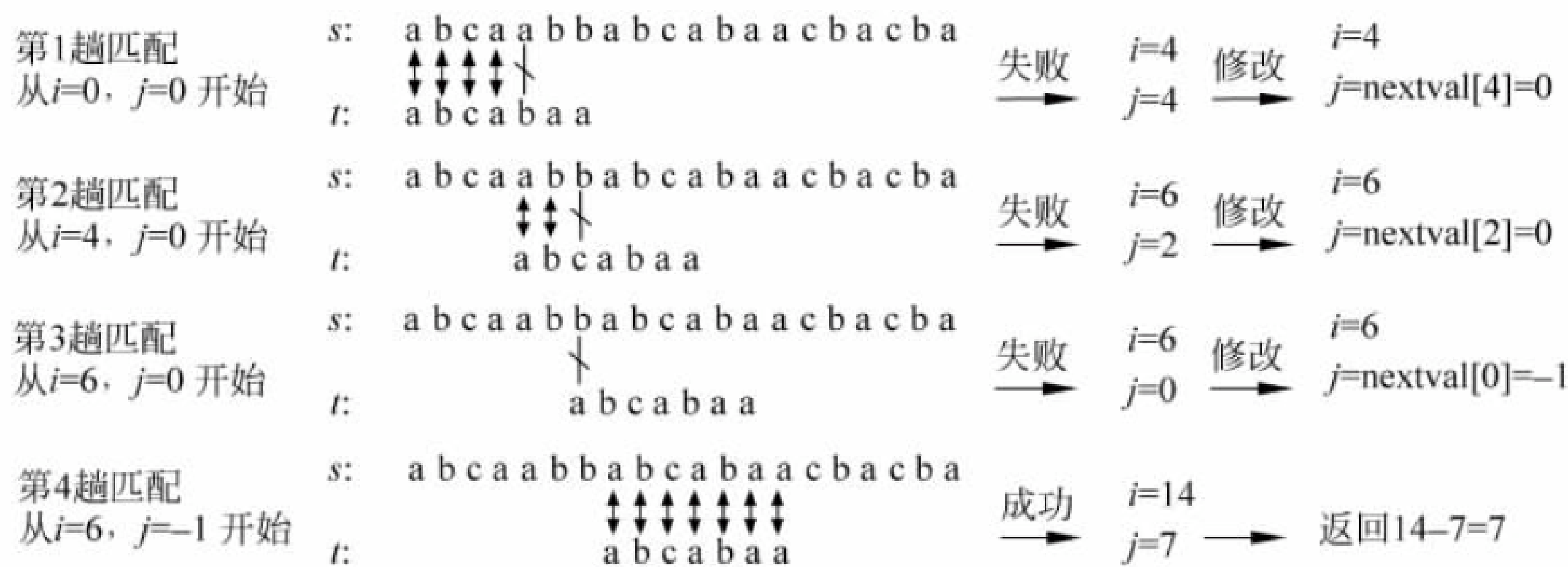


图 4.2 改进的 KMP 算法模式匹配的过程

(3) 从上述匹配过程看出:第 1 趟到第 4 趟的字符比较次数分别是 5、3、1、7,所以总共进行了 16 次字符比较。

7. 有两个顺序串 $s1$ 和 $s2$,设计一个算法求顺序串 $s3$,该串中的字符是 $s1$ 和 $s2$ 中的公共字符(即两个串都包含的字符)。

解:扫描 $s1$,对于当前字符 $s1$.data[i],若它在 $s2$ 中出现,则将其加入到串 $s3$ 中,最后返回 $s3$ 串。对应的算法如下:

```
SqString CommChar(SqString s1,SqString s2)
{   SqString s3;
```

```
        int i,j,k = 0;
        for (i = 0;i < s1.length;i++)
        {   for (j = 0;j < s2.length;j++)
                if (s2.data[j] == s1.data[i])
                    break;
            if (j < s2.length)          //s1.data[i]是公共字符
            {   s3.data[k] = s1.data[i];
                k++;
            }
        }
        s3.length = k;
        return s3;
    }
```

8. 采用顺序结构存储串，设计一个实现串通配符匹配的算法 pattern_index()，其中的通配符只有'?'，它可以和任何一个字符匹配成功。例如，pattern_index("? re","there are")返回的结果是 2。

解：采用 BF 算法的穷举法的思路只需要增加对'?'字符的处理功能。对应的算法如下：

```
int index(SqString s,SqString t)
{   int i = 0,j = 0;
    while (i < s.length && j < t.length)
    {   if (s.data[i] == t.data[j] || t.data[j] == '?')
        {   i++;
            j++;
        }
        else
        {   i = i - j + 1;
            j = 0;
        }
    }
    if (j >= t.length)
        return(i - t.length);
    else
        return( - 1);
}
```

9. 设计一个算法，在顺序串 s 中从后向前查找子串 t，即求 t 在 s 中最后一次出现的位置。

解：采用简单模式匹配算法。如果串 s 的长度小于串 t 的长度，直接返回 -1。然后 i 从 s.length$-t$.length 到 0 循环，再对于 i 的每次取值循环：置 $j=i$，$k=0$，若 s.data[j]$==t$.data[k]，则 j++，k++。循环中当 $k==t$.length 为真时，表示找到子串，返回物理下标 i。所有循环结束后都没有返回，表示串 t 不是串 s 的子串则返回 -1。对应的算法如下：

```
int LastPos1(SqString s,SqString t)
{   int i,j,k;
```

```
    if (s.length - t.length < 0)
        return -1;
    for (i = s.length - t.length; i >= 0; i--)
    {   for (j = i, k = 0; j < s.length && k < t.length && s.data[j] == t.data[k]; j++, k++);
        if (k == t.length)
            return i;
    }
    return -1;
}
```

10. 设计一个算法，判断一个字符串 s 是否为形如"序列1@为序列2"模式的字符序列，其中序列1和序列2都不含有'@'字符，且序列2是序列1的逆序列。例如"a+b@b+a"属于该模式的字符序列，而"1+3@3-1"不是。

解：建立一个临时栈 st 并初始化为空，其元素为 char 类型。置匹配标志 flag 为 true。扫描顺序串 s 的字符，将'@'之前的字符进栈。继续扫描顺序串 s 中'@'之后的字符，每扫描一个字符 e，退栈一个字符 x，若退栈时溢出或 e 不等于 x，则置 flag 为 false。循环结束后，若栈不空，置 flag 为 false。最后销毁栈 st 并返回 flag。对应的算法如下：

```
bool symm(SqString s)
{   int i = 0; char e, x;
    bool flag = true;
    SqStack * st;
    InitStack(st);
    while (i < s.length)            //将'@'之前的字符进栈
    {   e = s.data[i];
        if (e != '@')
            Push(st, e);
        else
            break;
        i++;
    }
    i++;                            //跳过@字符
    while (i < s.length && flag)
    {   e = s.data[i];
        if (!Pop(st, x)) flag = false;
        if (e != x) flag = false;
        i++;
    }
    if (!StackEmpty(st)) flag = false;
    DestroyStack(st);
    return flag;
}
```

11. 采用顺序结构存储串，设计一个算法求串 s 中出现的第一个最长重复子串的下标和长度。

解：采用简单模式匹配算法的思路，先给最长重复子串的起始下标 maxi 和长度 maxlen 均赋值为0。用 i 扫描串 s，对于当前字符 s_i，判定其后是否有相同的字符，若有记为

s_j，再判定 s_{i+1} 是否等于 s_{j+1}，s_{i+2} 是否等于 s_{j+2}，…，直到找到一个不同的字符为止，即找到一个重复出现的子串，把其起始下标 i 与长度 len 记下来，将 len 与 maxlen 相比较，保留较长的子串 maxi 和 maxlen。再从 s_{j+len} 之后查找重复子串。然后对于 s_{i+1} 之后的字符采用上述过程。循环结束后，maxi 与 maxlen 保存最长重复子串的起始下标与长度，将其复制到串 t 中。对应的算法如下：

```
void maxsubstr(SqString s,SqString &t)
{   int maxi = 0,maxlen = 0,len,i,j,k;
    i = 0;
    while (i < s.length)                //从下标为 i 的字符开始
    {   j = i + 1;                      //从 i 的下一个位置开始找重复子串
        while (j < s.length)
        {   if (s.data[i] == s.data[j])  //找一个子串,其起始下标为 i、长度为 len
            {   len = 1;
                for(k = 1;s.data[i + k] == s.data[j + k];k++)
                    len++;
                if (len > maxlen)       //将较大长度者赋给 maxi 与 maxlen
                {   maxi = i;
                    maxlen = len;
                }
                j += len;
            }
            else j++;
        }
        i++;                            //继续扫描第 i 字符之后的字符
    }
    t.length = maxlen;                  //将最长重复子串赋给 t
    for (i = 0;i < maxlen;i++)
        t.data[i] = s.data[maxi + i];
}
```

12. 用带头结点的单链表表示链串，每个结点存放一个字符。设计一个算法，将链串 s 中所有值为 x 的字符删除。要求算法的时间复杂度均为 $O(n)$、空间复杂度为 $O(1)$。

解：让 pre 指向链串头结点，p 指向首结点。当 p 不为空时循环：当 p－＞data＝＝x 时，通过 pre 结点删除 p 结点，再让 p 指向 pre 结点的后继结点；否则让 pre、p 同步后移一个结点。对应的算法如下：

```
void deleteall(LinkStrNode *&s,char x)
{   LinkStrNode *pre = s, *p = s -> next;
    while (p!= NULL)
    {   if (p -> data == x)
        {   pre -> next = p -> next;
            free(p);
            p = pre -> next;
        }
        else
        {   pre = p;                //pre、p 同步后移
```

```
            p = p -> next;
        }
    }
}
```

4.3 补充练习题及参考答案



4.3.1 单项选择题

1. 串是一种特殊的线性表，其特殊性体现在________。

A. 可以顺序存储　　B. 数据元素是一个字符

C. 可以链式存储　　D. 数据元素可以是多个字符

答：串中的每个数据元素只有一个字符。本题的答案为B。

2. 以下关于串的叙述中正确的是________。

A. 串是一种特殊的线性表　　B. 串中元素只能是字母

C. 空串就是空白串　　D. 串的长度必须大于零

答：A。

3. 串的长度是________。

A. 串中不同字母的个数　　B. 串中不同字符的个数

C. 串中所含字符的个数，且大于0　　D. 串中所含字符的个数

答：D。

4. 两个字符串相等的条件是________。

A. 串的长度相等　　B. 含有相同的字符集

C. 都是非空串　　D. 串的长度相等且对应的字符相同

答：D。

5. 设 S 为一个长度为 n 的字符串，其中的字符各不相同，则 S 中的互异非平凡子串（非空且不同于 S 本身）的个数为________。

A. 2^{n-1}　　B. $\frac{n(n+1)}{2}$　　C. $\frac{n(n+1)}{2}-1$　　D. $\frac{n(n-1)}{2}-1$

答：S 串中任意个连续的字符构成的子序列为其子串。S 串中长度为 $n-1$ 的子串个数为2，长度为 $n-2$ 的子串个数为3，…，长度为1的子串个数为 n。以上各种长度的子串组成了串 S 的互异非平凡子串集合，其个数为 $2+3+\cdots+n=\frac{n(n+1)}{2}-1$。本题的答案为C。

6. 若串 S="software"，其子串个数是________。

A. 8　　B. 37　　C. 36　　D. 9

答：由上题分析可知，长度为 n 的字符串（其中的字符各不相同）中含本身和空串的所有子串个数为 $\frac{n(n+1)}{2}+1$。这里 $n=8$，有37个子串。本题的答案为B。

7. 一个链串的结点类型如下：

```
typedef struct node
{   char data[MaxSize];
    struct node * next;
} SLinkNode;
```

如果每个字符占一个字节，结点大小为 6，指针占两个字节，该链串的存储密度为________。

A. 1/3　　B. 1/2　　C. 2/3　　D. 3/4

答：对于每个结点，其中存储的 6 个字符占 6 个字节，指针域占两个字节，所以存储密度=6/(6+2)=3/4。本题的答案为 D。

8. 串采用结点大小为 1 的链表作为其存储结构是指________。

A. 链表的长度为 1

B. 链表中只存放一个字符

C. 链表中每个结点的数据域中只存放一个字符

D. 以上都不对

答：C。

9. 设有两个串 s 和 t，判断 t 是否为 s 子串的算法称为________。

A. 求子串　　B. 串联接　　C. 串匹配　　D. 求串长

答：C。

10. 在 BF 模式匹配算法中，当模式串位 j 与目标串位 i 比较时两字符不相等，则 i 的位移方式是________。

A. i++　　B. $i=j+1$　　C. $i=i-j+1$　　D. $i=j-i+1$

答：C。

11. 在 BF 模式匹配算法中，当模式串位 j 与目标串位 i 比较时两字符不相等，则 j 的位移方式是________。

A. j++　　B. $j=0$　　C. $j=i-j+1$　　D. $j=j-i+1$

答：B。

12. 在 KMP 模式匹配中用 next 数组存放模式串的部分匹配信息，当模式串位 j 与目标串位 i 比较时两字符不相等，则 i 的位移方式是________。

A. i=next[j]　　B. i 不变　　C. $i=0$　　D. $i=i-j+1$

答：B。

13. 在 KMP 模式匹配中用 next 数组存放模式串的部分匹配信息，当模式串位 j 与目标串位 i 比较时两字符不相等，则 j 的位移方式是________。

A. $j=0$　　B. j=next[i]　　C. j 不变　　D. j=next[j]

答：D。

14. 在 KMP 模式匹配中用 next 数组存放模式串的部分匹配信息，当模式串位 j 与目标串 i 比较时两字符相等，则 i 的位移方式是________。

A. i++　　B. $i=j+1$　　C. $i=i-j+1$　　D. $i=j-i+1$

答：A。

15. 在 KMP 模式匹配中用 next 数组存放模式串的部分匹配信息，当模式串位 j 与目标串 i 比较时两字符相等，则 j 的位移方式是________。

A. j++　　B. $j=i+1$　　C. $j=i-j+1$　　D. $j=\text{next}[j]$

答：A。

16. 设目标串为 s、模式串为是 t，在 KMP 模式匹配中，next[4]=2 的含义是________。

A. 表示目标串匹配失败的位置是 $i=4$

B. 表示模式串匹配失败的位置是 $j=2$

C. 表示 t_4字符前面最多有两个字符和开头的两个字符相同

D. 表示 s_4字符前面最多有两个字符和开头的两个字符相同

答：C。

4.3.2 填空题

1. 串是指________。

答：含 $n(n\geqslant 0)$个字符的有限序列。

2. 设串 s1="I □am □a □student"，则串长为________。

答：空格也计算在串的长度之中，所以本题的答案为 14。

3. 字符串中任意个________称为该串的子串。

答：连续的字符组成的子序列。

4. 对于含有 n 个字符的顺序串 s，查找序号为 i 的字符，对应的时间复杂度为________。

答：$O(1)$。

5. 设目标串 s="abccdcdccbaa"，模式串 t="cdcc"，若采用 BF 模式匹配算法，则在第________趟匹配成功。

答：6。

6. 若 n 为主串长度，m 为子串长度，采用 BF 模式匹配算法，在最好的情况下需要的字符比较次数为________。

答：m。

7. 若 n 为主串长度，m 为子串长度，采用 BF 模式匹配算法，在最坏的情况下需要的字符比较次数为________。

答：$(n-m+1)\times m$。在采用 BF 算法时，最坏的情况下是子串恰好在主串的尾部，且前面每次都比较完子串的 m 个字符。

8. 已知模式串 t="aaababcaabbcc"，则 $t[3]$='b'，next[3]=________。

答：2。

9. 已知 t="abcaabbcabcaabdab"，该模式串的 next 数组值为________。

答：−1,0,0,0,1,1,2,0,0,1,2,3,4,5,6,0,1。

10. 已知模式串 t="ababaaab"，则 nextval 为________。

答：−1,0,−1,0,−1,3,1,0。

4.3.3 判断题

1. 判断以下叙述的正确性。

(1) 串中的每个元素只能是字母。

(2) 空串是只含有空格的串。

(3) 空串的长度为0。

(4) 含有 n 个字符的串中所有子串的个数为 $n(n+1)/2+1$。

(5) 如果一个串中的所有字符均在另一个串中出现,那么说明前者是后者的子串。

(6) 顺序串采用一个字符数组存放串中元素,所以顺序串等于一个字符数组。

答:

(1) 错误。

(2) 错误。

(3) 正确。

(4) 错误。只有在 n 个字符互不相同时命题才成立。

(5) 错误。

(6) 错误。

2. 判断以下叙述的正确性。

(1) KMP算法的最大特点是指示主串的指针不需回溯。

(2) 串的模式匹配算法有BF算法和KMP算法。在任何情况下KMP算法的时间性能都优于简单匹配算法。

(3) 模式串 t[0..11]为"aaababcaabbc",t[3]='b',则next[3]=1。

(4) 模式串 t[0..11]为"aaababcaabbc",t[2]='a',则nextval[2]=-1。

(5) 改进的KMP算法除将next改为nextval以外,它们的匹配过程是一样的。

答:(1)正确。

(2) 错误。例如,s="abcd",t="123"时,KMP算法比BF算法的执行时间多。

(3) 错误。next[3]=2。

(4) 正确。

(5) 正确。

4.3.4 简答题

1. C/C++语言中提供了字符串的一组功能函数,为什么在数据结构中还要讨论串?

答:主要有两个目的,一是将串作为一种数据结构,体现从逻辑结构→存储结构→运算的数据结构的观点;二是通过讨论串算法设计,使读者掌握串运算算法的实现细节。

2. 空串与空格串有何区别?

答:空串是指不含任何字符的串,其长度为0,它是任意串的子串。仅含有空格字符的串称为空格串,其长度为串中空格字符的个数。

3. 给定两个串 s_1 和 s_2,判断 s_2 是否为 s_1 旋转而来的,例如"waterbottle"是"erbottlewat"旋转而来的。问只使用isSubString(用于子串判断)和Concat(串连接)算法能否完成该功能?

答：可以。设 s_1="erbottlewat"=xy(x="erbottle",y="wat"),则 s_2="waterbottle"=yx。yx 肯定是 $xyxy$ 的子串,即 $s_2=yx$ 一定是 $xyxy=s_1s_1$的子串。

所以,如果 isSubString(s_2,Concat(s_1,s_1))为真,则 s_2是 s_1旋转而来的;否则,s_2不是 s_1旋转而来的。

4. 如果模式串中没有任何加速匹配的信息,此时 KMP 算法和 BF 算法执行的趟数是相同的。如果你认为正确,请说明;如果你认为不正确,请给出一个反例。

答:如果模式串中没有任何加速匹配的信息,此时 KMP 算法和 BF 算法执行的趟数不一定相同。例如,目标串 s="abcabcd",模式串 t="abcd",采用 BF 算法时的匹配过程如下:

第1趟匹配(从i=0,j=0开始)　s: abcabcd　t: abcd　i=3 j=3 修改→ i=i−j+1=1 j=0

第2趟匹配(从i=1,j=0开始)　s: abcabcd　t: abcd　i=1 j=0 修改→ i=i−j+1=2 j=0

第3趟匹配(从i=2,j=0开始)　s: abcabcd　t: abcd　i=2 j=0 修改→ i=i−j+1=3 j=0

第4趟匹配(从i=3,j=0开始)　s: abcabcd　t: abcd　i=7 j=4 成功→ 返回 i−4=3

共 4 趟,所需要的字符比较次数=4+1+1+4=10。

如果采用 KMP 算法,模式串有 next[0]=−1,next[0]=next[1]=next[2]=0。其匹配过程如下:

第1趟匹配(从i=0,j=0开始)　s: abcabcd　t: abcd　i=3 j=3 修改→ i=3 j=next[3]=0

第2趟匹配(从i=3,j=0开始)　s: abcabcd　t: abcd　i=7 j=4 成功→ 返回 i−4=3

共两趟,所需要的字符比较次数=4+4=8。

5. 并非在任何情况下 KMP 算法都好于 BF 算法,请给出一个 BF 算法好于 KMP 算法的例子。

答:例如目标串 s="abcdefghijk",模式串 t="efg",KMP 算法几乎退化为 BF 算法,但 KMP 算法还要花费时间计算 next 数组。

6. 若主串 s="abcaabccacabcabcaaaabc",模式串 t="abcabcaaa"。

(1) 求出 t 的 next 数组。

(2) 给出采用 KMP 算法求子串位置的过程。

(3) 总共进行了多少次字符比较?

答:(1) 对于模式串 t="abcabcaaa",先计算 next 数组,其结果如表 4.5 所示,计算过

程如下。

当 $j=0$ 时，next[0]=−1(固定值)；

当 $j=1$ 时，next[1]=0(固定值)；

当 $j=2$ 时，$t_0 \neq t_1$，next[2]=0；

当 $j=3$ 时，$t_0 \neq t_2$，next[3]=0；

当 $j=4$ 时，$t_0=t_3$="a"，$k=1$，next[4]=k=1；

当 $j=5$ 时，$t_0t_1=t_3t_4$="ab"，$k=2$，next[5]=k=2；

当 $j=6$ 时，$t_0t_1t_2=t_3t_4t_5$="abc"，$k=3$，next[6]=k=3；

当 $j=7$ 时，$t_0t_1t_2t_3=t_3t_4t_5t_6$="abca"，$k=4$，next[7]=$k$=4；

当 $j=8$ 时，$t_0t_1 \neq t_6t_7$，$t_0=t_7$="a"，$k=1$，next[8]=k=1。

表 4.5　模式串 t 对应的 next 数组

j	0	1	2	3	4	5	6	7	8
$t[j]$	a	b	c	a	b	c	a	a	a
next[j]	−1	0	0	0	1	2	3	4	1

(2) 采用 KMP 算法求子串位置的过程如下(开始时 $i=0$，$j=0$)：

```
第1趟匹配        s: abcaabccacabcabcaaaabc      i=4   修改   i=4
(从i=0，j=0开始)                                j=4   ──→   j=next[4]=1
                 t: abcabcaaa

第2趟匹配        s: abcaabccacabcabcaaaabc      i=4   修改   i=4
(从i=4，j=1开始)                                j=1   ──→   j=next[1]=0
                 t:    abcabcaaa

第3趟匹配        s: abcaabccacabcabcaaaabc      i=7   修改   i=7
(从i=4，j=0开始)                                j=3   ──→   j=next[3]=0
                 t:     abcabcaaa

第4趟匹配        s: abcaabccacabcabcaaaabc      i=7   修改   i=7
(从i=7，j=0开始)                                j=0   ──→   j=next[0]=-1
                 t:        abcabcaaa

第5趟匹配        s: abcaabccacabcabcaaaabc      i=9   修改   i=9
(从i=7，j=-1开始)                               j=1   ──→   j=next[1]=0
                 t:         abcabcaaa

第6趟匹配        s: abcaabccacabcabcaaaabc      i=9   修改   i=9
(从i=9，j=0开始)                                j=0   ──→   j=next[0]=-1
                 t:          abcabcaaa

第7趟匹配        s: abcaabccacabcabcaaaabc      i=19  成功   返回
(从i=9，j=-1开始)                               j=9   ──→   i-9=10
                 t:           abcabcaaa
```

(3) 从上述匹配过程看出：第 1 趟到第 7 趟的字符比较次数分别是 5、1、4、1、2、1、9，所以总共进行了 23 次字符比较。

7. 已知 KMP 串匹配算法中模式串 t 为"babababaa"，给出 next 数组改进后的 nextval 数组。

答：先求出 t 的 next 数组，然后求改进后的 nextval 数组，如表 4.6 所示。

表 4.6 模式串对应的 next 数组和 nextval 数组

j	0	1	2	3	4	5	6	7	8
$t[j]$	b	a	b	a	b	a	b	a	a
next$[j]$	−1	0	0	1	2	3	4	5	6
nextval$[j]$	−1	0	−1	0	−1	0	−1	0	6

8. 设目标为 s="abcaabbcaaabababaabca"，模式为 t="babab"。

(1) 计算模式串 t 的 nextval 数组值。

(2) 不写算法，画出利用改进 KMP 算法进行模式匹配的过程。

(3) 总共进行了多少次字符比较？

答：(1) 先计算 next 数组，在此基础上求 nextval 数组，如表 4.7 所示。

表 4.7 计算 next 数组和 nextval 数组

j	0	1	2	3	4
$t[j]$	b	a	b	a	b
next$[j]$	−1	0	0	1	2
nextval$[j]$	−1	0	−1	0	−1

(2) 改进 KMP 算法进行模式匹配的过程如下：

第1趟匹配 (从i=0，j=0开始)
s: abcaabbcaaabababaabca
t: babab
i=0，j=0 修改→ i=0，j=nextval[0]=−1

第2趟匹配 (从i=0，j=−1开始)
s: abcaabbcaaabababaabca
t: babab
i=2，j=1 修改→ i=2，j=nextval[1]=0

第3趟匹配 (从i=2，j=0开始)
s: abcaabbcaaabababaabca
t: babab
i=2，j=0 修改→ i=2，j=nextval[0]=−1

第4趟匹配 (从i=2，j=−1开始)
s: abcaabbcaaabababaabca
t: babab
i=3，j=0 修改→ i=3，j=nextval[0]=−1

第5趟匹配 (从i=3，j=−1开始)
s: abcaabbcaaabababaabca
t: babab
i=4，j=0 修改→ i=4，j=nextval[0]=−1

第6趟匹配 (从i=4，j=−1开始)
s: abcaabbcaaabababaabca
t: babab
i=6，j=1 修改→ i=6，j=nextval[1]=0

第7趟匹配 (从i=6，j=0开始)
s: abcaabbcaaabababaabca
t: babab
i=7，j=1 修改→ i=7，j=nextval[1]=0

趟次	匹配	修改前		修改后
第8趟匹配 (从i=7，j=0开始)	s: abcaabbcaaabababaabca t: babab	i=7 j=0	修改→	i=7 j=nextval[0]=−1
第9趟匹配 (从i=7，j=0开始)	s: abcaabbcaaabababaabca t: babab	i=8 j=0	修改→	i=8 j=nextval[0]=−1
第10趟匹配 (从i=8，j=−1开始)	s: abcaabbcaaabababaabca t: babab	i=9 j=0	修改→	i=9 j=nextval[0]=−1
第11趟匹配 (从i=9，j=−1开始)	s: abcaabbcaaabababaabca t: babab	i=10 j=0	修改→	i=10 j=nextval[0]=−1
第12趟匹配 (从i=10，j=−1开始)	s: abcaabbcaaabababaabca t: babab	i=16 j=5	修改→	返回 i−5=11

(3) 从上述匹配过程看出：第1趟到第12趟的字符比较次数分别是1、2、1、1、1、2、2、1、1、1、1、5，所以总共进行了19次字符比较。尽管匹配的趟数比较多，但字符的比较次数并不多。

4.3.5 算法设计题

1.【顺序串算法】设计一个高效算法，将顺序串的所有字符逆置，要求算法的空间复杂度为$O(1)$。

解：扫描顺序串s的前半部分元素，对于元素s.data[i]($0 \leqslant i < s$.length/2)，将其与后半部分的对应元素s.data[s.length−i−1]进行交换。对应的算法如下：

```
void reverse(SqString &s)
{   for (int i = 0;i < s.length/2;i++)
        swap(s.data[i],s.data[s.length - i - 1]);
                 //data[i]与 data[L.length - i - 1]交换
}
```

2.【顺序串算法】对于采用顺序结构存储的串，设计一个比较这两个串是否相等的算法Equal()。

解：两个串相等是指长度相等且对应位置的字符必须都相同。先比较两串的长，在相等时扫描两串，逐一比较相应位置的字符，若相同继续比较直到全部比较完毕，如果都相同则表示两串相等，否则表示两串不相等。对应的算法如下：

```
bool Equal(SqString s,SqString t)
{   int i = 0;
    bool flag = true;
    if (s.length!= t.length)
```

```
        return false;
    else
    {   while (i<s.length && flag)
        {   if (s.data[i]!=t.data[i])
                flag=false;
            i++;
        }
        return flag;
    }
}
```

3.【顺序串算法】设计一个算法，将顺序串 s 中所有值为 $c1$ 的字符换成 $c2$ 的字符。

解：从头到尾扫描 s 串，将值为 $c1$ 的元素直接替换成 $c2$ 即可。对应的算法如下：

```
void Trans(SqString &s,char c1,char c2)
{   int i;
    for (i=0;i<s.length;i++)
        if (s.data[i]==c1)
            s.data[i]=c2;
}
```

4.【顺序串算法】设计一个算法，从顺序串 s 中删除值等于 c 的所有字符。

解：从头到尾扫描 s 串，对于值为 c 的元素采用移动的方式进行删除。算法如下：

```
void DelAll(SqString &s,char c)
{   int i,j;
    for (i=0;i<s.length;i++)
        if (s.data[i]==c)
        {   for (j=i;j<s.length;j++)
                s.data[j]=s.data[j+1];
            s.length--;
        }
}
```

上述算法效率很低，更高效的算法如下（思路参见《教程》例 2.3 的算法二）：

```
void DelAll1(SqString &s,char c)
{   int k=0,i=0;                        //k 记录值等于 c 的字符个数
    while (i<s.length)
    {   if (s.data[i]==c)
            k++;
        else
            s.data[i-k]=s.data[i];      //当前字符前移 k 个位置
        i++;
    }
    s.length-=k;                        //串 s 的长度递减
}
```

5.【顺序串算法】设计一个算法，从顺序串 s 中第 index 个字符起求出首次与字符串 t 相同的子串的起始位置。

解：采用 BF 算法思路，从第 index 个元素开始扫描 s，当其元素值与 t 的第一个元素的值相同时判定它们之后的元素值是否依次相同，直到 t 结束为止。若都相同则返回，否则继续上述过程直到 s 扫描完为止。对应的算法如下：

```
int PartPos(SqString s,SqString t,int index)    //s 为主串,t 为子串
{   int i,j,k;
    int n = s.length;
    int m = t.length;
    for (i = index;i <= n - m;i++)
    {   for (j = 0,k = i;j < m &&t.data[j] == s.data[k];k++,j++);
        if (j == m)
            return(i);
    }
    return( - 1);
}
```

6.【顺序串算法】采用顺序结构存储串，设计一个算法计算指定子串在一个字符串中出现的次数，如果该子串不出现则为 0。

解：本题是 BF 模式匹配算法的扩展，在 s 中找到子串 t 后不是退出，而是继续查找，直到整个字符串查找完毕。对应的算法如下：

```
int substrcount(SqString s,SqString t)
{   int i = 0,j,k,count = 0;
    for (i = 0;s.data[i];i++)
    {   for (j = i,k = 0;j < s.length && k < t.length &&
            (s.data[j] == t.data[k]);j++,k++);
        if (k >= t.length)
            count++;
    }
    return(count);
}
```

7.【顺序串算法】采用顺序结构存储串，设计一个算法，求串 s 和串 t 的一个最长公共子串。

解：以 s 为主串，t 为子串，设 maxidx 为最长公共子串在 s 中的序号，maxlen 为最长公共子串的长度。采用 BF 算法扫描串 s 和扫描串 t，当 s 的当前字符等于 t 的当前字符时比较后面的字符是否相等，这样得到一个公共子串(其在 s 中的起始位置为 i，长度为 len)。将 len 与 maxlen 相比，若 len 较大，则置 maxlen＝len，maxidx＝i。如此直到扫描完 s 为止。对应的算法如下：

```
SqString MaxComStr(SqString s,SqString t)
{   SqString str;                 //str 用于存放最长公共子串
    int maxidx = 0,maxlen = 0,i,j,k,len;
    i = 0;                        //i 作为扫描 s 的指针
    while (i < s.length)
    {   j = 0;                    //j 作为扫描 t 的指针
        while (j < t.length)
        {   if (s.data[i] == t.data[j])
            {   len = 1;          //找一个公共子串,其在 s 中的位置为 i,长度为 len
                for (k = 1;i + k < s.length && j + k < t.length
                        && s.data[i + k] == t.data[j + k];k++)
                    len++;
                if (len > maxlen)   //将较大长度者赋给 idx 与 len
                {   maxidx = i;
                    maxlen = len;
                }
                j += len;         //继续扫描 t 中第 j + len 字符之后的字符
            }
            else j++;
        }
        i++;                      //继续扫描 s 中第 i 字符之后的字符
    }
    for (i = 0;i < maxlen;i++)
        str.data[i] = s.data[maxidx + i];
    str.length = maxlen;
    return(str);                  //返回最长公共子串
}
```

8.【顺序串算法】设计一个程序,计算顺序串 s 中每一个字符出现的次数。

解:设计一个结构体数组 cnum 用于存放顺序串 s 中出现的字符和出现的次数。用 i 扫描 s,用 k 记录 cnum 中的元素个数,对于 s.data[i],若在 cnum 数组中没有对应字符,将 s.data[i]直接放到 cnum 中,否则将对应字符的出现次数增 1。对应的程序如下:

```
#include "sqstring.cpp"         //包含顺序串基本运算算法
typedef struct
{   char c;                     //字符
    int num;                    //字符计数
} CType;
int fun(SqString s,CType cnum[])
{   int i,j,k = 0;              //k 记录 cnum 中的元素个数
    for (i = 0;i < s.length;i++)
    {   if (k == 0)             //cnum 中没有元素时将 s.data[i]直接放到 cnum 中
        {   cnum[k].c = s.data[i];
            cnum[k].num = 1;
            k++;
        }
        else                    //cnum 中存在元素时查找是否有相同的字符
```

```
        {   for (j = 0;j < k && s.data[i]!= cnum[j].c;j++);
            if (j >= k)             //s.data[i]放入 cnum 数组中
            {   cnum[k].c = s.data[i];
                cnum[k].num = 1;
                k++;
            }
            else cnum[j].num++;
        }
    }
    return k;
}
int main()
{   int i,k;
    char str[MaxSize];
    SqString s;
    CType cnum[MaxSize];
    printf("输入串:"); gets(str);
    StrAssign(s,str);
    printf("统计结果如下:\n");
    k = fun(s,cnum);
    for (i = 0;i < k;i++)
        printf("   %c %d\n",cnum[i].c,cnum[i].num);
    return 1;
}
```

9.【链串算法】设计一个算法 Equal(),在链串上实现判断两个串是否相等的功能。

解:扫描两个链串,并同步比较当前结点值是否相等,若不相等返回 false;否则继续比较到结束,若均相等且均结束,则返回 true。对应的算法如下:

```
bool Equal(LinkStrNode *s,LinkStrNode *t)
{   LinkStrNode *p = s, *q = t;
    while (p!= NULL && q!= NULL &&flag)
    {   if (p->data!= q->data)
            return false;
        p = p->next;
        q = q->next;
    }
    if (p!= NULL || q!= NULL)
        return false;
    else
        return true;
}
```

10.【链串算法】假设采用链串存储结构,设计一个算法,在链串 s 中找子串 t 最后出现的首字符的序号(逻辑序号,序号从 1 开始),如果串 t 不是串 s 的子串,返回 0。

解:采用 BF 模式匹配算法的思路。idx 用于求子串 t 在 s 中最后出现的首字符的序号,其初值为 0。用 p 扫描链串 s,i 记录结点 p 的逻辑序号,初始值为 0,当 p 不为 NULL 时循环:i 增 1,$q=p$,r 指向串 t 的首结点,当两结点的值相同时循环,即 q、r 指针均后移一

个结点，如果 r 为 NULL，表示找到了一个子串，置 idx$=i$。循环结束，最后返回 idx。对应的算法如下：

```
int LastPos(LinkStrNode *s,LinkStrNode *t)
{   int i=0,idx=0;
    LinkStrNode *p=s->next, *q, *r;
    while (p!=NULL)
    {   i++;
        q=p;
        r=t->next;
        while (q!=NULL && r!=NULL && q->data==r->data)
        {   q=q->next;
            r=r->next;
        }
        if (r==NULL)            //找到子串
            idx=i;
        p=p->next;
    }
    return idx;
}
```

第5章 递归





5.1 本章知识体系

1. 知识结构图

本章的知识结构如图 5.1 所示。

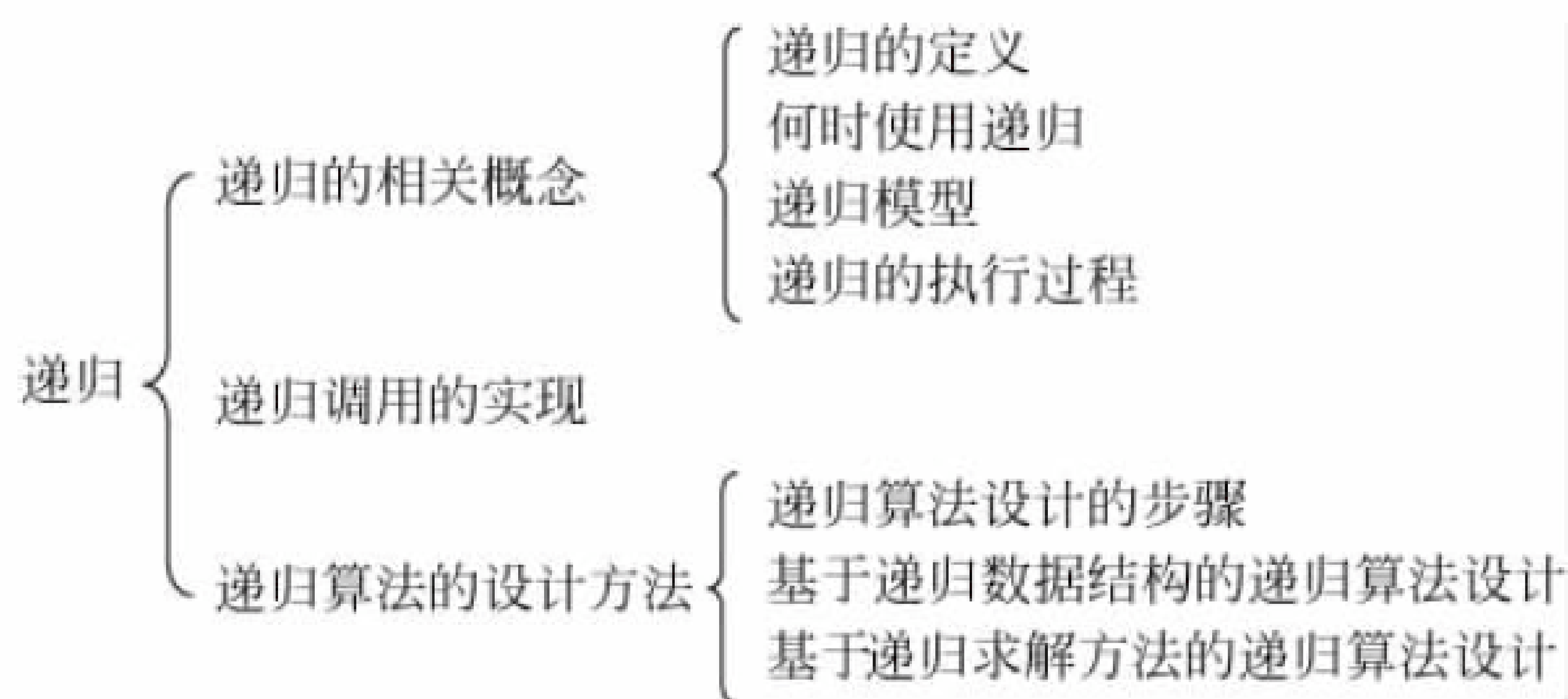


图 5.1 第 5 章知识结构图

2. 基本知识点

(1) 何时使用递归。

(2) 如何从递归角度提取求解问题的递归模型。

(3) 递归算法的执行过程。

(4) 递归算法的实现原理。

(5) 递归算法设计的一般步骤。

(6) 理解递归数据结构的特征。

(7) 利用递归思想求解复杂的应用问题。

3. 要点归纳

(1) 递归分为直接递归和间接递归,而间接递归算法都可以转换为直接递归算法来实现。

(2) 在递归算法中递归调用语句是最后一条执行语句时称为尾递归。

(3) 如果求解问题的定义是递归的、存放数据的数据结构是递归的或者问题的求解方法是递归的,一般使用递归算法来求解。

(4) 递归模型是递归算法的抽象,它反映一个递归问题的递归结构。在设计递归算法时首先获取求解问题的递归模型,然后转换为相应的递归算法。

(5) 递归模型由递归出口和递归体两部分构成。递归出口确定递归到何时结束,而递归体确定递归求解时的递推关系。

(6) 递归思路是把一个不能或不好直接求解的"大问题"转化成一个或几个"小问题"来解决。

(7) 函数调用是通过一个栈来实现的,用于保存返回地址、函数实参和局部变量值等。

(8) 一般情况下,尾递归算法可以通过循环或者迭代方式转换为等价的非递归算法。对于不是尾递归的复杂递归算法,可以用栈来模拟递归执行过程,从而将其转换为等价的非

递归算法。

(9) 获取递归模型的通常分为3个步骤,即分析问题、提取递归体和提取递归出口。在实际中需要根据求解问题来操作。

5.2 教材中的练习题及参考答案

1. 有以下递归函数:

```
void fun(int n)
{   if (n == 1)
        printf("a: % d\n",n);
    else
    {   printf("b: % d\n",n);
        fun(n - 1);
        printf("c: % d\n",n);
    }
}
```

分析调用 fun(5)的输出结果。

解: 在调用递归函数 fun(5)时先递推到递归出口,然后求值。这里的递归出口语句是 printf("a:%d\n",n),递推时执行的语句是 printf("b:%d\n",n),求值时执行的语句是 printf("c:%d\n",n)。调用 fun(5)的输出结果如下:

```
b:5
b:4
b:3
b:2
a:1
c:2
c:3
c:4
c:5
```

2. 已知 $A[0..n-1]$为整数数组,设计一个递归算法求这 n 个元素的平均值。

解: 设 avg(A,i)返回 $A[0..i]$共 $i+1$ 个元素的平均值,则递归模型如下。

$$\text{avg}(A,i)=\begin{cases}A[0] & \text{当 } i=0\\ (\text{avg}(A,i-1)*i+A[i])/(i+1) & \text{其他情况}\end{cases}$$

对应的递归算法如下:

```
float avg(int A[],int i)
{   if (i == 0)
        return(A[0]);
```

```
    else
        return((avg(A,i-1) * i + A[i])/(i + 1));
}
```

求 $A[n]$中 n 个元素平均值的调用方式为 avg(A,$n-1$)。

3. 设计一个算法求正整数 n 的位数。

解：设 $f(n)$为整数 n 的位数，其递归模型如下。

$$f(n)=\begin{cases}1 & \text{当 } n<10 \text{ 时}\\ f(n/10)+1 & \text{其他情况}\end{cases}$$

对应的递归算法如下：

```
int fun(int n)
{   if (n < 10)
        return 1;
    else
        return fun(n/10) + 1;
}
```

4. 上楼可以一步上一阶，也可以一步上两阶，设计一个递归算法，计算共有多少种不同的走法。

解：设 $f(n)$表示 n 阶楼梯的不同的走法数，显然 $f(1)=1$，$f(2)=2$(2 阶有一步一步走和两步走两种走法)。$f(n-1)$表示 $n-1$ 阶楼梯的不同的走法数，$f(n-2)$表示 $n-2$ 阶楼梯的不同的走法数，对于 n 阶楼梯，第 1 步上一阶有 $f(n-1)$种走法，第 1 步上两阶有 $f(n-2)$种走法，则 $f(n)= f(n-1)+ f(n-2)$。对应的递归算法如下：

```
int fun(int n)
{   if (n == 1 || n == 2)
        return n;
    else
        return fun(n - 1) + fun(n - 2);
}
```

5. 设计一个递归算法，利用顺序串的基本运算求串 s 的逆串。

解：经分析，求逆串的递归模型如下。

$$f(s)=\begin{cases}s & \text{若 } s=\Phi\\ \text{Concat}(f(\text{SubStr}(s,2,\text{StrLength}(s)-1)),\text{SubStr}(s,1,1)) & \text{其他情况}\end{cases}$$

递归思路是：对于 $s=$"$s_1s_2\cdots s_n$"的串，假设"$s_2s_3\cdots s_n$"已求出其逆串，即 f(SubStr(s,2,StrLength(s)-1))，再将 s_1(为 SubStr(s,1,1))单个字符构成的串连接到最后即得到 s 的逆串。对应的递归算法如下：

```
#include "sqstring.cpp"            //顺序串的基本运算算法
SqString invert(SqString s)
```

```
{   SqString s1,s2;
    if (StrLength(s)>0)
    {   s1=invert(SubStr(s,2,StrLength(s)-1));
        s2=Concat(s1,SubStr(s,1,1));
    }
    else
        StrCopy(s2,s);
    return s2;
}
```

6. 设有一个不带表头结点的单链表 L，设计一个递归算法 count(L)求以 L 为首结点指针的单链表的结点个数。

解：对应的递归算法如下。

```
int count(LinkNode *L)
{   if (L==NULL)
        return 0;
    else
        return count(L->next)+1;
}
```

7. 设有一个不带表头结点的单链表 L，设计两个递归算法，traverse(L)正向输出单链表 L 的所有结点值，traverseR(L)反向输出单链表 L 的所有结点值。

解：对应的递归算法如下。

```
void traverse(LinkNode *L)
{   if (L==NULL) return;
    printf("%d ",L->data);
    traverse(L->next);
}
void traverseR(LinkNode *L)
{   if (L==NULL) return;
    traverseR(L->next);
    printf("%d ",L->data);
}
```

8. 设有一个不带表头结点的单链表 L，设计两个递归算法，del(L,x)删除单链表 L 中第一个值为 x 的结点，delall(L,x)删除单链表 L 中所有值为 x 的结点。

解：对应的递归算法如下。

```
void del(LinkNode *&L,ElemType x)
{   LinkNode *t;
    if (L==NULL) return;
    if (L->data==x)
    {   t=L;L=L->next;free(t);
        return;
    }
```

```
        del(L->next,x);
    }
    void delall(LinkNode *&L,ElemType x)
    {   LinkNode *t;
        if (L==NULL) return;
        if (L->data==x)
        {   t=L;L=L->next;
            free(t);
        }
        delall(L->next,x);
    }
```

9. 设有一个不带表头结点的单链表 L，设计两个递归算法，maxnode(L)返回单链表 L 中的最大结点值，minnodel(L)返回单链表 L 中的最小结点值。

解：对应的递归算法如下。

```
ElemType maxnode(LinkNode *L)
{   ElemType max;
    if (L->next==NULL)
        return L->data;
    max=maxnode(L->next);
    if (max>L->data) return max;
    else return L->data;
}
ElemType minnode(LinkNode *L)
{   ElemType min;
    if (L->next==NULL)
        return L->data;
    min=minnode(L->next);
    if (min>L->data) return L->data;
    else return min;
}
```

10. 设计一个模式匹配算法，其中模板串 t 含有通配符"*"，它可以和任意子串匹配。对于目标串 s，求其中匹配模板 t 的一个子串的位置("*"不能出现在 t 的最开头和末尾)。

解：采用 BF 模式匹配的思路，当是 $s[i]$和 $t[j]$比较，而 $t[j]$为'*'时，取出 s 中对应'*'的字符之后的所有字符构成的字符串，即 SubStr(s,$i+2$,s.length$-i-1$)，其中 $i+2$ 是 s 中对应'*'字符后面一个字符的逻辑序号。再取出 t 中'*'字符后面的所有字符构成的字符串，即 SubStr(t,$j+2$,t.length$-j-1$)，递归对它们进行匹配，若返回值大于-1，表示匹配成功，返回 i，否则返回-1。对应的递归算法如下：

```
#include "sqstring.cpp"          //顺序串的基本运算算法
findpat(SqString s,SqString t)
{   int i=0,j=0,k;
    while (i<s.length && j<t.length)
    {   if (t.data[j]=='*')
        {   k=findpat(SubStr(s,i+2,s.length-i-1),SubStr(t,j+2,t.length-j-1));
```

```
                j++;
                if (k>-1)
                    return i-1;
                else
                    return -1;
            }
            else if (s.data[i] == t.data[j])
            {   i++;
                j++;
            }
            else
            {   i=i-j+1;
                j=0;
            }
        }
        if (j>=t.length)
            return i-1;
        else
            return -1;
}
```

5.3 补充练习题及参考答案 

5.3.1 单项选择题

1. 一个正确的递归算法通常包含________。

   A. 递归出口　　B. 递归体

   C. 递归出口和递归体　　D. 以上都不包含

答：C。

2. 递归函数 $f(1)=1, f(n)=f(n-1)+n(n>1)$ 的递归出口是________。

   A. $f(1)=1$　　B. $f(1)=0$　　C. $f(0)=0$　　D. $f(n)=n$

答：当 $n=1$ 时，$f(1)=1$。本题的答案为 A。

3. 递归函数 $f(1)=1, f(n)=f(n-1)+n(n>1)$ 的递归体是________。

   A. $f(1)=1$　　B. $f(0)=0$

   C. $f(n)=f(n-1)+n$　　D. $f(n)=n$

答：递归体反映递归过程。本题的答案为 C。

4. 计算 $f(n)=\frac{1}{1\times2}+\frac{1}{2\times3}+\cdots+\frac{1}{n(n+1)}$，采用的递归模型为________。

   A. $f(n)=\frac{1}{1\times2}+\frac{1}{2\times3}+\cdots+\frac{1}{n(n+1)}$

   B. $f(n)=\frac{1}{1\times2}+\frac{1}{2\times3}+\cdots+\frac{1}{(n-1)n}+f(n-1)$

C. $f(1)=\frac{1}{2}, f(n)=f(n-1)+\frac{1}{n(n+1)}$

D. $f(n)=f(n-1)+\frac{1}{n(n+1)}$

答：递归模型由两部分组成，一个是递归出口，另一个是递归体，在上述选项中只有C符合条件。本题的答案为C。

5. 在将递归算法转换成对应的非递归算法时，通常需要使用________保存中间结果。

A. 队列　　B. 栈　　C. 链表　　D. 树

答：B。

6. 函数 $f(x,y)$ 定义如下：

$$f(x,y)=\begin{cases} f(x-1,y)+f(x,y-1) & \text{当 } x>0 \text{ 且 } y>0 \\ x+y & \text{否则} \end{cases}$$

则 $f(2,1)$ 的值是________。

A. 1　　B. 2　　C. 3　　D. 4

答：D。计算过程为 $f(2,1)=f(1,1)+f(2,0)=f(0,1)+f(1,0)+2=1+1+2=4$。

7. 一个递归问题可以用递归算法求解，也可以用非递归算法求解，但单从执行时间来看，通常递归算法比非递归算法________。

A. 较快　　B. 较慢　　C. 相同　　D. 无法比较

答：B。

8. 以下关于递归的叙述中错误的是________。

A. 一般而言，使用递归解决问题较使用循环解决问题需要定义更多的变量

B. 递归算法的执行效率相对较低

C. 递归算法的执行需要用到系统栈

D. 以上都是错误的

答：A。

9. 在系统实现递归调用时需利用递归工作记录保存参数值。在传值参数情形，需为对应形参分配空间，以存放实参的________。

A. 空间　　B. 副本　　C. 代码地址　　D. 地址

答：B。

10. 在系统实现递归调用时需利用递归工作记录保存参数值。对于引用型参数，需保存实参的________，在被调用程序中需直接操纵实参。

A. 空间　　B. 副本　　C. 值　　D. 地址

答：D。

5.3.2 填空题

1. 将 $f(n)=1+\frac{1}{2}+\frac{1}{3}+\cdots+\frac{1}{n}$ 转化成递归函数，其递归出口是____①____，递归体是____②____。

答：① $f(1)=1$　② $f(n)=f(n-1)+\frac{1}{n}(n\geqslant 2)$。

2. 有以下递归过程：

```
void print(int w)
{    int i;
     if (w!=1)
     {    print(w-1);
          for (i=0;i<=w;i++)
               printf("%3d",w);
          printf("\n");
     }
}
```

调用语句 print(4)的结果是________。

答：

```
1
2  2
3  3  3
4  4  4  4
```

3. 有以下递归过程：

```
void reverse(int m)
{    printf("%d",n%10);
     if (n/10 != 0)
          reverse(n/10);
}
```

调用语句 reverse(582)的结果是________。

答：285。reverse(m)将整数 m 的各数字位逆置。

4. 递推式 $f(1)=0, f(n)=f(n/2)+1$ 的解是________。

答：$\log_2 n$。不妨设 $n=2^k$，有 $k=\log_2 n$，则 $f(n)=f(n/2)+1=1+f(n/2)=1+(1+f(n/2^2))=2+f(n/2^2)=\cdots=k+\text{fun}(n/2^k)=k+f(1)=\log_2 n$。

5. 设 $a[0..n-1]$是一个含有 n 个整数的数组，求该数组中所有元素之和的递归定义是________。

答：$f(a,0)=a[0], f(a,i)=a[i]+f(a,i-1)\ (i\geqslant 1)$。

6. 有以下递归算法：

```
void fun(int n)
{    if (n>0)
     {    printf("%d ",n);
          fun(n-1);
          fun(n-1);
     }
}
```

执行 fun(3)的输出是________。

答：3 2 1 1 2 1 1。

7. 有以下递归算法：

```
void fun(int n)
{    if (n>0)
     {    fun(n-1);
          fun(n-1);
          printf("%d ",n);
     }
}
```

执行 fun(3)的输出是________。

答：1 1 2 1 1 2 3。

5.3.3 判断题

1. 判断以下叙述的正确性。

(1) 任何递归算法都有递归出口。

(2) 递归算法的执行效率比功能相同的非递归算法的执行效率高。

(3) 任何能够正确执行的递归算法都能转换为功能等价的非递归算法。

(4) 任何递归算法都是尾递归。

(5) 通常递归的算法简单、易懂、容易编写，而且执行的效率也高。

(6) 尾递归算法可以通过循环转换成非递归算法。

答：(1) 正确。

(2) 错误。通常情况下递归算法的执行效率比功能相同的非递归算法的执行效率低。

(3) 正确。

(4) 错误。

(5) 错误。

(6) 正确。

2. 判断以下叙述的正确性。

(1) 有以下递归算法：

```
int fun(int n)
{    if (n==1 || n==0)
          return n;
     else
          return n+fun(n/2);
}
```

其中递归体是 $n==1$ 或 $n==0$ 时返回 n。

(2) 有以下递归函数：

```
int fun(int n)
{    if (n==1 || n==0)
```

```
        return n;
    else
        return n + fun(n - 2);
}
```

执行 fun(6)的返回结果是 10。

答：(1) 错误。

(2) 错误。执行 fun(6)的返回结果是 12。

5.3.4 简答题

1. 简述递归算法的优缺点。

答：递归算法的优点是结构清晰、可读性强，而且容易用数学归纳法来证明算法的正确性，因此它为设计算法、调试程序带来了很大的方便。

递归算法的缺点是算法的运行效率较低，无论是耗费的计算时间还是占用的存储空间都比功能等价的非递归算法要多。

2. 推导出求 x 的 n 次幂的递归模型。

答：设 $f(x,n)=x^n$，$f(x,n-1)=x^{n-1}$，所以 $f(x,n)=x*x^{n-1}=n*f(x,n-1)$。对应的递归模型如下：

$$f(x,n)=\begin{cases} x & \text{当 } n=1 \text{ 时} \\ n*f(x,n-1) & \text{当 } n>1 \text{ 时} \end{cases}$$

3. 推导出求 x 的 n 次幂的递归模型，要求最多使用 $O(\log_2 n)$次递归调用。

答：设 $f(x,n)=x^n$，$f(x,n/2)=x^{n/2}$，当 n 为偶数时，$f(x,n)=x^{n/2}*x^{n/2}=f(x,n/2)*f(x,n/2)$；当 n 为奇数时，$f(x,n)=x*x^{(n-1)/2}*x^{(n-1)/2}=x*f(x,(n-1)/2)*f(x,(n-1)/2)$。对应的递归模型如下：

$$f(x,n)=\begin{cases} x & \text{当 } n=1 \text{ 时} \\ f(x,n/2)*f(x,n/2) & \text{当 } n \text{ 为大于 1 的偶数时} \\ x*f(x,(n-1)/2)*f(x,(n-1)/2) & \text{当 } n \text{ 为大于 1 的奇数时} \end{cases}$$

4. 采用递归方法，将含有 n 个字符的 C/C++字符串 s 中最后一个为 x 的字符改为 y，给出其递归模型。

答：对应的递归模型如下：

$$f(s,n,x,y)\equiv\begin{cases} \text{不做任何事情} & \text{当 } n=0 \text{ 时} \\ s[n-1]=y & \text{当 } s[n-1]=x \text{ 时} \\ f(s,n-1,x,y) & \text{其他情况} \end{cases}$$

5. 某递归算法的求解时间复杂度的递推式如下：

$$T(n)=\begin{cases} 1 & \text{当 } n=0 \\ T(n-1)+n+3 & \text{当 } n>0 \end{cases}$$

求该算法的时间复杂度。

答：$T(n)=T(n-1)+(n+3)=T(n-2)+(n+2)+(n+3)$

$=T(n-3)+(n+1)+(n+2)+(n+3)$

$=\cdots$

$=T(0)+4+5+\cdots+(n+1)+(n+2)+(n+3)$

$=1+4+5+\cdots+(n+1)+(n+2)+(n+3)$

$=n(n+7)/2+1$

$=O(n^2)$

6. 设 $a[0..n-1]$是含有 n 个元素的整数数组，写出求 n 个整数之积的递归定义。

答：设 $f(a,i)$返回 $a[0..i-1]$共 i 个元素之积。当 $i=1$，有 $f(a,1)=a[0]$。假设数组 a 的前 $i-1$ 个元素之积已求出，即已知 $f(a,i-1)$，则 $f(a,i)=f(a,i-1)*a[i-1]$，所以得到以下递归定义：

$f(a,1)=a[0]$

$f(a,i)=f(a,i-1)*a[i-1]$　当 $i>1$

a 中 n 个整数之积 $=f(a,n)$。

7. 以下算法是计算两个正整数 u 和 v 最大公因数的递归函数，给出其递归模型。

```
int gcd(int u,int v)
{   int r;
    if ((r = u % v) == 0)
        return(v);
    else
        return(gcd(u,r));
}
```

答：由上述函数得到以下递归模型。

$$\gcd(u,v)=\begin{cases} v & \text{当 } u\%v=0 \text{ 时} \\ \gcd(u,u\%v) & \text{其他情况} \end{cases}$$

5.3.5　算法设计题

1. 【递归算法设计】有一个不带头结点的单链表，其结点类型为 LinkNode。设计一个递归算法，删除并释放以 h 为首指针的单链表中的所有结点。

解：设 h 为不带头结点的单链表(含 n 个结点)，则 h->next 也是一个不带头结点的单链表(含 $n-1$ 个结点)，两者除了相差一个结点以外，其他都是相似的。设 release(h)的功能是删除并释放单链表 h 中的所有结点。其递归模型如下：

$$\text{release}(h)\equiv\begin{cases} \text{不做任何事情} & \text{当 } h \text{ 为空表} \\ \text{release}(h\text{->next})\text{；释放 } h \text{ 结点} & \text{其他情况} \end{cases}$$

对应的递归算法如下：

```
void release(LinkNode *h)
{   if (h!= NULL)
    {   release(h->next);
        free(h);                //释放 h 结点
    }
}
```

2. 【递归算法设计】设计一个递归算法，利用串的基本运算 SubStr()判断字符 x 是否

在串 s 中。

解：设串 s="$a_1a_2\cdots a_n$"，设 Find(s,x)的值表示 x 是否为串 s 的元素，若是则返回真，否则返回假。本题的递归模型如下：

$$\text{Find}(s,x)=\begin{cases}0 & \text{如果 s 为空串}\\ 1 & \text{如果 } a_1=x\\ \text{Find}(\text{"}a_2\cdots a_n\text{"},x) & \text{其他情况}\end{cases}$$

对应的递归算法如下：

```
#include "SqString.cpp"                    //包含顺序串的定义和基本运算函数
bool Find(SqString s,char x)
{   SqString s1;
    if (s.length==0)
        return false;
    else if (s.data[0]==x)                  //a1=x
        return true;
    else
    {   s1=SubStr(s,2,s.length-1);          //s1="a2…an"
        return(Find(s1,x));
    }
}
```

3.【递归算法设计】一个人赶着鸭子去每个村庄卖，每经过一个村子卖去所赶鸭子的一半又一只，这样他经过了 7 个村子后还剩两只鸭子。设计一个算法求他出发时共赶了多少只鸭子？

解：设 fun(i)表示经过 i 个村子后还剩下的鸭子数，依题意有以下递归模型。

$$\text{fun}(i)=\begin{cases}2 & \text{当 } i=7 \text{ 时}\\ 2*\text{fun}(i+1)+1 & \text{当 } i<7 \text{ 时}\end{cases}$$

对应的递归算法如下：

```
int fun(int i)
{   if (i==7)
        return 2;
    else
        return 2*fun(i+1)+1;
}
```

调用 fun(0)求得出发时共赶鸭子数为 383 只。

4.【递归算法设计】求解猴子吃桃问题。海滩上有一堆桃子，5 只猴子来分。第一只猴子把这堆桃子分为 5 份，多了一个，这只猴子把多的一个扔入海中，拿走了一份。第 2 只猴子把剩下的桃子又平均分成 5 份，又多了一个，它同样把多的一个扔入海中，拿走了一份，第 3、第 4、第 5 只猴子都是这样做的，问海滩上原来最少有多少个桃子？

解：设 fun(i)表示第 i 个猴子分桃子前的桃子总数。显然，第 5 只猴子分桃子后的桃子总数为 m(相当于第 6 个猴子分桃子前的桃子总数，可以是任何大于等于 0 的整数)。$f(n+1)$ 应该是$(f(n)-1)/5$ 的 4 倍，即 $f(n+1)=4[(f(n)-1)/5]$，求出 $f(n)=5f(n+1)/4+1$，

而 $f(n)$一定为整数，所以 m 应该取保证所有 $5f(n+1)/4$ 整除的最小整数。依题意有以下递归模型：

fun(6)= m　　　　　　　　　第 5 只猴子分桃子后的桃子总数为 m

fun(n)=(fun($n+1$)+1) * 5　　　当 $n>1$

对应的递归算法如下：

```
bool isn(int x,int y)        //x 整除 y 时返回 true
{   if (x % y==0)
        return true;
    else
        return false;
}
int fun(int n,int m)
{   if (n==6)
        return m;
    else
    {   if (isn(5 * fun(n+1,m),4))
            return (5 * fun(n+1,m)/4+1);
        else                 //当 m 不合适时返回 -1
            return -1;
    }
}
int pnumber()
{   int k;
    int m=0;                 //m 从 0 开始试探
    while(true)
    {   k=fun(1,m);
        if (k!= -1)
            break;
        m++;
    }
    return k;
}
```

pnumber()的计算结果是 3121，所以海滩上原来最少有 3121 个桃子。

5.【递归算法设计】有以下递归计算公式：

$$C(n,0)=1 \qquad n\geqslant 0$$

$$C(n,n)=1 \qquad n\geqslant 0$$

$$C(n,m)=C(n-1,m)+C(n-1,m-1) \qquad n>m,n\geqslant 0,m\geqslant 0$$

设计一个递归算法和一个非递归算法求 $C(n,m)$。

解：对应的递归算法如下。

```
int fun(int n,int m)
{   if (n>=0 && m==0 || n>=0 && m==n)
        return 1;
    else
```

```
    {   if (n>m && n>=0 && m>=0)
            return(fun(n-1,m) + fun(n-1,m-1));
        else
        {   printf("n,m 值不正确\n");
            return(-1);
        }
    }
}
```

用一个数组 a 存放 $C(m,n)$的值，对应的非递归算法如下：

```
int fun1(int n, int m)
{   int a[M][N] = {0},i,j;
    for (i=0;i<=n;i++)
    {   a[i][0] = 1;
        a[i][i] = 1;
    }
    for (j=1;j<=m;j++)
        for (i=j+1;i<=n;i++)
            a[i][j] = a[i-1][j] + a[i-1][j-1];
    return a[n][m];
}
```

6. 【递归算法设计】编写一个递归算法，读入一个字符串(以“.”作为结束)，要求打印出它们的倒序字符串。

解：首先获取用户按键，如果不是'.'字符，则递归调用该过程，否则显示该字符。对应的算法如下：

```
void reverse()
{   char  ch;
    scanf("%c",&ch);
    if (ch!= '.')
    {   reverse();
        printf("%c",ch);
    }
}
```

7. 【递归算法设计】设计一个程序求解全排列问题：输入 n 个不同的字符，给出它们所有的 n 个字符的全排列。

解：将 n 个不同的字符存放在字符串 str[0..$n-1$]中，设 $f(\text{str},k,n)$表示输出 str[k..$n-1$](共 $n-k$ 个字符)所有字符全排列，而 $f(\text{str},k+1,n)$表示输出 str[$k+1$..$n-1$](共 $n-k-1$ 个字符)所有字符全排列，前者是大问题，后者为小问题。递归模型 $f(\text{str},k,n)$如下：

$$f(\text{str},k,n)\equiv\begin{cases}\text{输出产生的解} & \text{若 } k=n-1\\ \text{对于 } k\sim n-1 \text{ 的 } i\text{，str}[i]\text{与 str}[k]\text{交换位置；} & \text{其他情况}\\ \quad f(\text{str},k+1,n)\text{；} & \\ \quad \text{将 str}[k]\text{与 str}[i]\text{交换位置(恢复环境)；} & \end{cases}$$

对应的算法如下：

```
void print(char str[],int n)        //输出一个排列
{    for (int i=0;i<n;i++)
         printf(" %c ",str[i]);
     printf("\n");
}
void perm(char str[],int k,int n)
{    int i;
     if (k==n-1)
         print(str,n);
     else
     {    for (i=k;i<n;i++)
          {    swap(str[k],str[i]);   //交换 str[k]与 str[i]
               perm(str,k+1,n);
               swap(str[k],str[i]);   //交换 str[k]与 str[i]
          }
     }
}
```

设计以下主函数：

```
int main()
{    int n=3;
     char a[4]="123";
     printf("123 的全排列如下:\n");
     perm(a,0,n);
     return 1;
}
```

程序的执行结果如下：

```
123 的全排列如下:
1 2 3
1 3 2
2 1 3
2 3 1
3 2 1
3 1 2
```

8.【递归算法设计】设计一个程序求解组合问题：从自然数 $1\sim n$ 中任取 r 个数的所有组合。

解：设 comb(a,m,k)为从 $1\sim m$ 中任取 k 个数的所有组合（每个组合放在数组 a 中，由于组合与元素顺序无关，不妨设 $a[0]<a[1]<a[2]<\cdots$）。当组合的最后一个数字（只能取 $k\sim m$ 中的一个数）选定后，其后的数字是从余下的 $m-1$ 个数中取 $k-1$ 个数的组合。求解组合问题的递归模型如下：

$$\text{comb}(a,m,k) \equiv \begin{cases} \text{输出产生的解} & \text{若 } k=1 \\ \text{对于 } k \sim m \text{ 的 } i, a[k-1]=i; & \text{其他情况} \\ \quad \text{comb}(a,i-1,k-1); & \end{cases}$$

对应的程序如下：

```
#include <stdio.h>
#define MaxSize 10
int n,r;                              //全局变量
void print(int a[])                   //输出一个组合
{   int j;
    for (j=r-1;j>=0;j--)
        printf(" %d ",a[j]);
    printf("\n");
}
void comb(int a[],int m,int k)
{   int i;
    for (i=m;i>=k;i--)
    {   a[k-1]=i;
        if (k>1)
            comb(a,i-1,k-1);
        else
            print(a);
    }
}
int main()
{   int a[MaxSize];
    printf("输入 n,r(r<=n):");
    scanf("%d%d",&n,&r);
    printf("1..%d 中 %d 个的组合结果如下:\n",n,r);
    comb(a,n,r);
    printf("\n");
    return 1;
}
```

程序的一次执行结果如下：

```
输入 n,r(r<=n):5 3↙
1..5 中 3 个的组合结果如下:
5 4 3
5 4 2
5 4 1
5 3 2
5 3 1
5 2 1
4 3 2
4 3 1
4 2 1
3 2 1
```

9.【递归算法设计】棋子移动问题：有 $2n$ 个棋子（$n\geq4$）排成一行，开始位置为白色全部在左边，黑色全部在右边。移动棋子的规则是每次必须同时移动相邻两个棋子，颜色不限，可以左移也可以右移到空位上去，但不能调换两个棋子的左右位置。每次移动必须跳过若干个棋子（不能平移），要求最后能够移成黑白相间的一行棋子。

解：$n=4$ 的求解过程如下。

```
          1 2 3 4 5 6 7 8 9 10
初始状态：○○○○●●●●——
  第 1 步：○○○——●●●○●      // mvtosp(4)：即位置 4 开头的两个棋子与"--"交换
  第 2 步：○○○●○●●——●      // mvtosp(8)：即位置 8 开头的两个棋子与"--"交换
  第 3 步：○——●○●●○○●      // mvtosp(2)：即位置 2 开头的两个棋子与"--"交换
  第 4 步：○●○●○●——○●      // mvtosp(7)：即位置 7 开头的两个棋子与"--"交换
  第 5 步：——○●○●○●○●      // mvtosp(1)：即位置 1 开头的两个棋子与"--"交换
```

用数组 $c[1..2n+2]$ 存放棋子，用 init()函数对其所有元素初始化。设 mv(n)表示求解 $2n$ 个棋子移动的问题，则棋子移动问题求解的递归模型如下：

$$\mathrm{mv}(n)\equiv\begin{cases}\text{直接求解} & n=4\\ \text{将 c}[n]\text{、c}[n+1]\text{棋子对移动到 c}[2n+1]\text{、c}[2n+2]\text{处，即 mv}(n)\text{；} & \text{其他情况}\\ \quad\text{将 c}[2n-1]\text{、c}[2n]\text{棋子对移动到 c}[n]\text{、c}[n+1]\text{处，即 mv}(2n-1)\text{；} & \\ \quad\text{mv}(n-1)\text{；} & \end{cases}$$

对应的程序如下：

```
#include <stdio.h>
#include <string.h>
const int MAX = 100;
char c[MAX][3];
int st = 0,sp,n;                    //全局变量,st 记录移动的步骤,sp 指向为"-"的棋子
void print()                        //输出一个移动步骤
{   printf("  step % - 2d:",++st);
    for (int i = 1;i <= 2 * n + 2;i++)
        printf(" % s",c[i]);
    printf("\n");
}
void mvtosp(int k)                  //将 c[k]和 c[k + 1]两个棋子与"--"交换
{   for (int j = 0;j <= 1;j++)
    {   strcpy(c[sp + j],c[k + j]);
        strcpy(c[k + j],"—");
    }
    sp = k;                         //sp 指向"--"的棋子
    print();                        //输出一个解
}
void mv(int n)                      //求解 2n 个棋子移动问题
{   if (n == 4)                     //递归出口
    {   mvtosp(4);mvtosp(8);mvtosp(2);
        mvtosp(7);mvtosp(1);
    }
    else                            //求解 n > 4 的情况
```

```
        {   mvtosp(n);
            mvtosp(2 * n - 1);
            mv(n - 1);
        }
}
void init(int n)                    //初始化 c 数组
{   int i;
    st = 0;
    sp = 2 * n + 1;                 //sp 指向第 2n + 1 的棋子,即"--"
    for (i = 1;i <= n;i++)
        strcpy(c[i],"○");
    for (i = n + 1;i <= 2 * n;i++)
        strcpy(c[i],"●");
    strcpy(c[2 * n + 1],"—");
    strcpy(c[2 * n + 2],"—");
}
int main()
{   do
    {   printf("输入 n 值(4 - 20):");
        scanf("%d",&n);
    } while (n > 20 || n < 4);
    init(n);
    printf("移动过程如下:\n");
    mv(n);
    printf("\n");
    return 1;
}
```

本程序的一次执行结果如下：

```
输入 n 值(4 - 20):8↙
移动过程如下:
  step1 :○○○○○○○——●●●●●●●○●
  step2 :○○○○○○○●●●●●●●——○●
  step3 :○○○○○○——●●●●●●○●○●
  step4 :○○○○○○●●●●●●——○●○●
  step5 :○○○○○——●●●●●○●○●○●
  step6 :○○○○○●●●●●——○●○●○●
  step7 :○○○○——●●●●○●○●○●○●
  step8 :○○○○●●●●——○●○●○●○●
  step9 :○○○——●●●○●○●○●○●○●
  step10:○○○●○●●——●○●○●○●○●
  step11:○——●○●●○○●○●○●○●○●
  step12:○●○●○●——○●○●○●○●○●
  step13:——○●○●○●○●○●○●○●○●
```

10.【递归算法设计】设以字符序列 a、b、c、d 作为顺序栈 st 的输入，利用 push(进栈)和

pop(出栈)操作,输出所有可能的出栈序列并编程实现整个算法。

解:设 $A=(a_1,a_2,\cdots,a_m)$是已出栈的序列,$B=(b_1,b_2,\cdots,b_n)$是已进栈的序列(如果栈不空),$C=(c_1,c_2,\cdots,c_k)$是尚未进栈的序列,描述进栈、出栈的状态由 A、C 两个序列表示即可(由 A、C 序列可以确定 B 序列)。

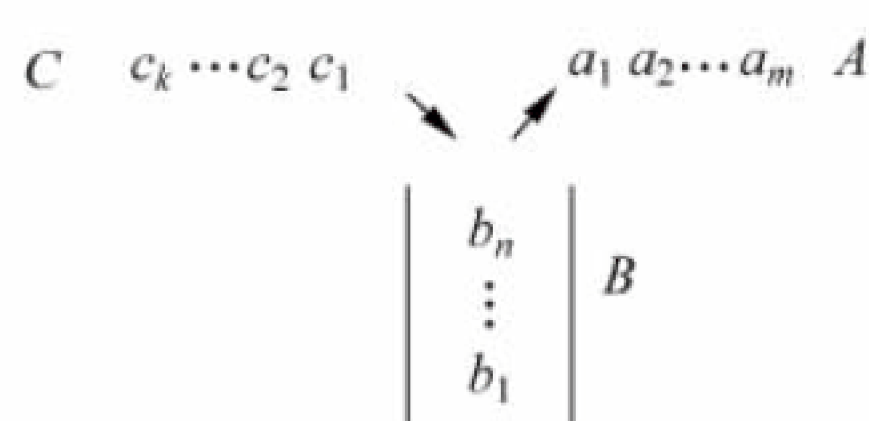


图 5.2　进栈、出栈的状态

产生所有出栈序列的过程是如果所有元素都在 A 序列中(栈空且 C 序列中的所有元素处理完),此时产生一种出栈序列,否则从某个进栈、出栈的状态(如图 5.2 所示的状态称为初始状态)出发,只有两种操作。

① 从初始状态出发,将 C 中的某元素(如 c_1)进栈,此时产生另一种进栈出栈的状态,从该状态继续进行类似的操作。

② 从初始状态出发,将 C 中的某元素出栈(如 b_n),即出栈元素 b_n并将其添加到 A 的末尾,此时产生另一种进栈、出栈的状态,从该状态继续进行类似的操作。

求解过程如图 5.3 所示,其中"从该状态继续进行类似的操作"和产生所有出栈序列的过程是相似的,所以可以采用递归算法。

注意:上述①、②两步都是从初始状态出发的,所以每步执行后都需要将进栈、出栈的状态恢复成初始状态,如第①步进栈了一个元素,其后要将该元素出栈以恢复成原来的初始状态,第②步出栈了一个元素 x,其后要将 x 进栈以恢复成原来的初始状态。

为了算法方便,用数组 str[0..total-1]表示一个进栈序列(其中元素个数为 total),每个下标对应唯一的元素,所以在进栈、出栈中直接用 1~n 的下标来表示,只有在最后输出一个出栈序列时还原为字符序列。用 a 数组表示输出序列,用 curp 表示 a 数组中当前元素的位置,st 是一个栈,包含存放元素的 data 数组和栈顶指针 top。

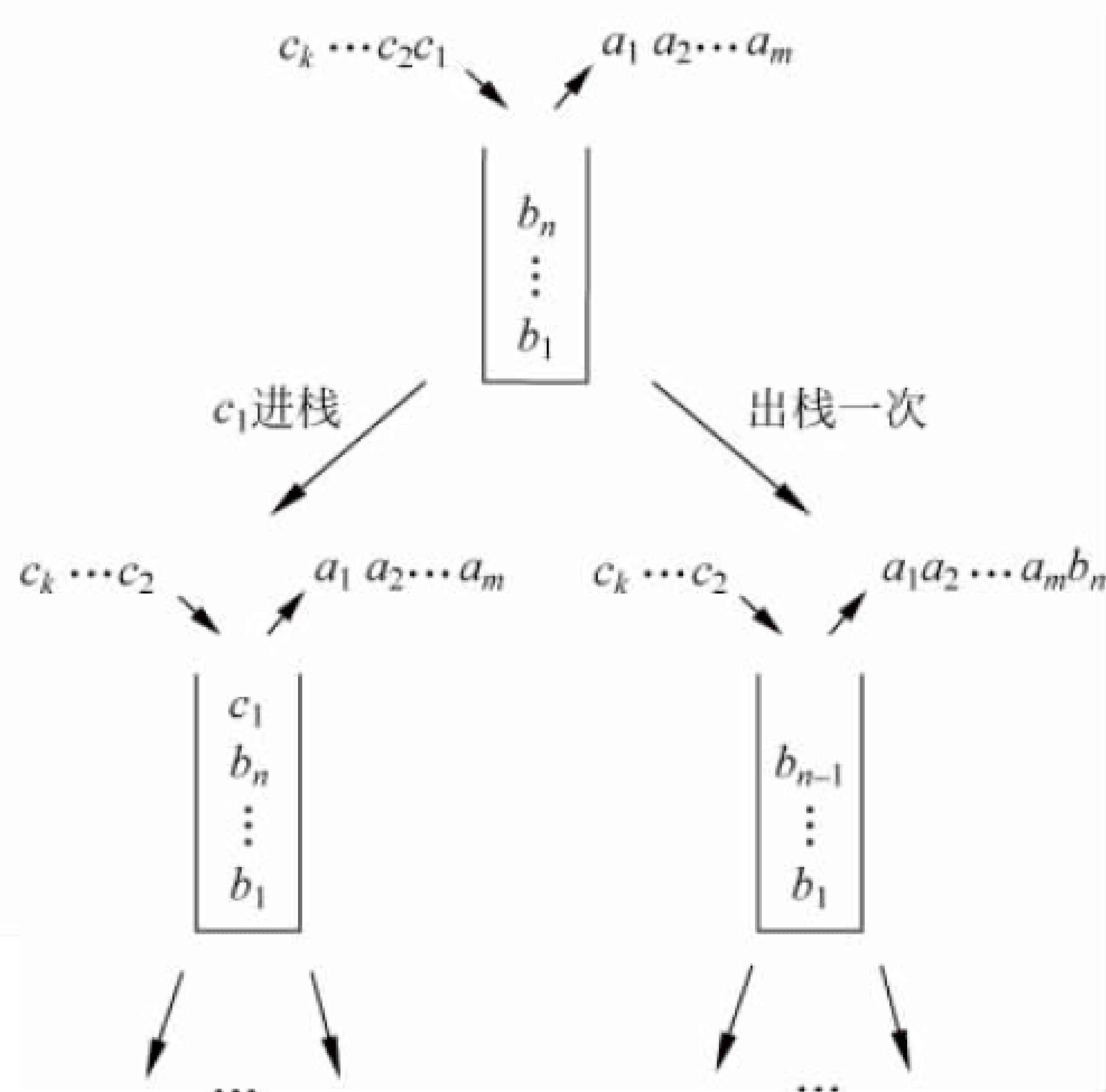


图 5.3　求解过程

前面介绍过,进栈、出栈的状态由 A、C 两个序列表示即可,A 序列中有 curp 个元素,C 序列的当前处理元素设为 m(表示 C 序列为 $c_k\cdots c_m$),这样进栈、出栈的状态由 m、curp 唯一

确定。为了简单，栈运算只设计了几个基本的处理过程。完整的程序如下：

```
#include <stdio.h>
#define MaxSize 100
struct stacknode
{    int data[MaxSize];
     int top;
} st;                                     //全局变量,定义整数顺序栈
int total = 4;                            //全局变量,定义输入序列的元素个数
char str[] = "abcd";                      //全局变量,指定进栈序列
int sum = 0;                              //全局变量,累计出栈序列个数
void initstack()                          //初始化顺序栈
{
     st.top = -1;
}
void push(int n)                          //元素 n 进栈运算
{    st.top++;
     st.data[st.top] = n;
}
int pop()                                 //退栈运算
{    int temp;
     temp = st.data[st.top];
     st.top--;
     return temp;
}
bool empty()                              //判断栈空否运算
{    if (st.top == -1)
         return true;
     else
         return false;
}
void process(int m,int a[],int curp)      //处理 C 序列中 m 位置的元素
{    int x,i;
     if (m > total && empty())            //输出一种可能的方案
     {   printf("   ");
         for (i = 0;i < curp;i++)         //输出 a 中的元素序列,构成一种出栈序列
             printf(" %c ",str[a[i] - 1]);
         printf("\n");
         sum++;                           //出栈序列个数增 1
     }
     if (m <= total)                      //编号 m 的元素进栈时递归
     {   push(m);                         //m 进栈
         process(m + 1,a,curp);           //递归：从该状态继续进行类似的操作
         pop();                           //出栈以恢复环境
     }
     if (!empty())                        //编号 m 的元素出栈时递归
     {   x = pop();                       //出栈 x
         a[curp] = x;                     //将 x 输出到 a 中
         curp++;                          //a 中元素个数增 1
         process(m,a,curp);               //递归：从该状态继续进行类似的操作
```

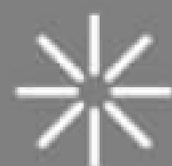
```
        push(x);                   //进栈以恢复环境
    }
}
int main()
{   int a[MaxSize];
    initstack();
    printf("所有出栈序列:\n");
    process(1,a,0);                //m 从 1 开始,curp 从 0 开始
    printf("出栈序列个数:%d\n",sum);
    return 1;
}
```

上述程序的执行结果如下：

```
所有出栈序列:
d c b a
c d b a
c b d a
c b a d
b d c a
b c d a
b c a d
b a d c
b a c d
a d c b
a c d b
a c b d
a b d c
a b c d
出栈序列个数:14
```

第6章 数组和广义表

6.1 本章知识体系



1. 知识结构图

本章的知识结构如图 6.1 所示。

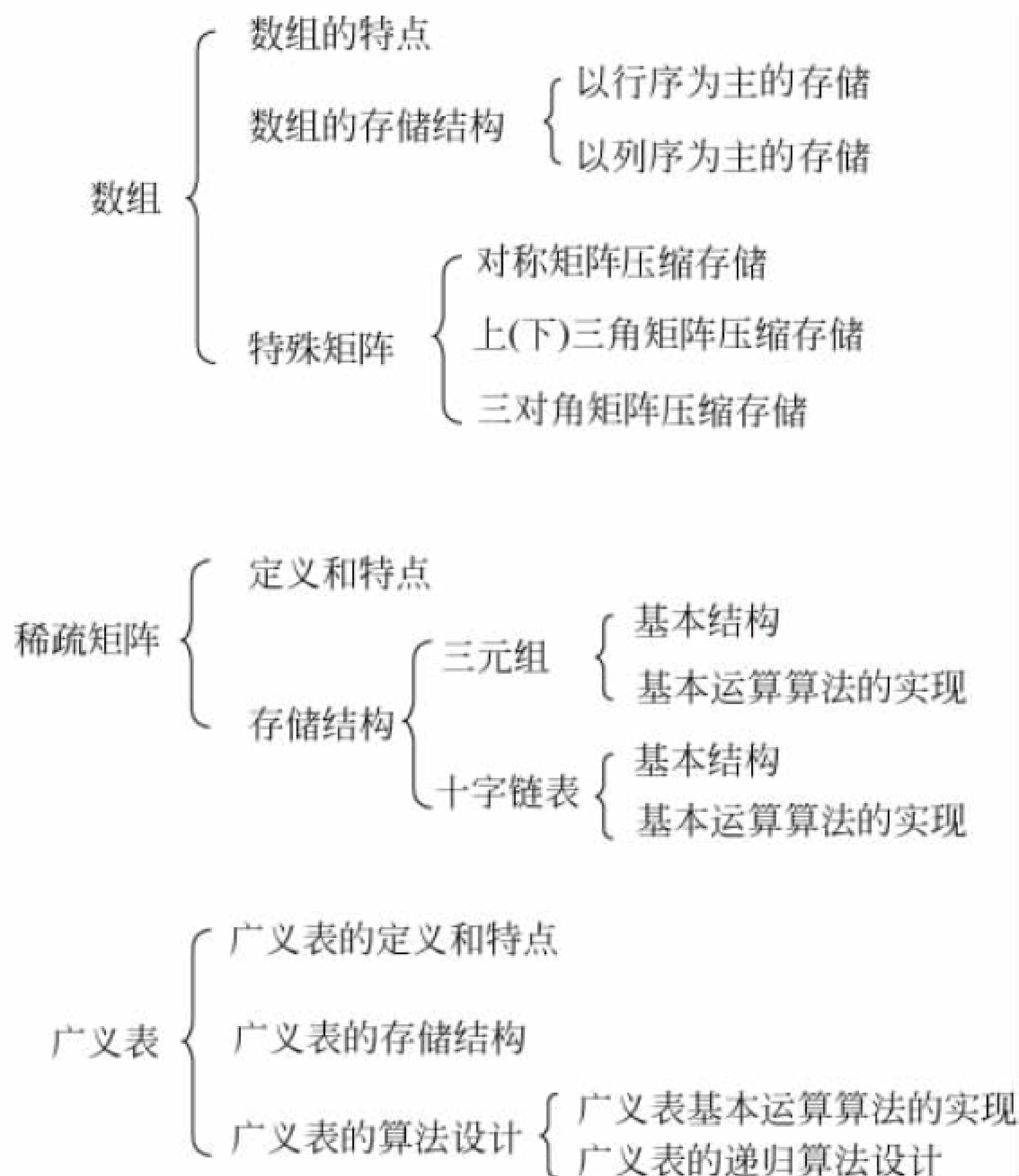


图 6.1　第 6 章知识结构图

2. 基本知识点

(1) 数组的顺序存储结构及其元素地址计算方法。

(2) 对称矩阵、上三角矩阵、下三角矩阵和三对角矩阵的压缩存储方法。

(3) 稀疏矩阵的三元组存储结构及其基本运算算法设计。

(4) 稀疏矩阵十字链表存储结构的特点。

(5) 广义表的定义和特点。

(6) 广义表的链式存储结构及其递归特性。

(7) 广义表的递归算法设计方法。

3. 要点归纳

(1) 数组是线性表的推广，$d(d\geqslant 1)$维数组中存在 d 个线性关系。其主要的操作是取指定位置的元素值和给指定位置的元素赋值。

(2) 数组通常采用顺序存储方法，分以行优先和以列优先两种存储方式。

(3) 数组采用顺序存储方法后具有随机存取特性。

(4) 特殊矩阵并不是指具有特殊用途的矩阵，是指一类元素值分布具有某种规律的矩阵，它们都是 $n \times n$ 的方阵，主要包括对称矩阵、上（下）三角矩阵和对角矩阵。

(5) 特殊矩阵的常规压缩存储方法是将重复值的元素或者特殊值的元素（如为某个常量）仅仅存储一次。通常将特殊矩阵 A 压缩存储到一个一维数组 B 中，A 中元素 $a_{i,j}$ 对应 B 中元素 b_k，$k=f(i,j)$，f 为地址变换函数。

(6) 特殊矩阵采用压缩存储的目的是节省存储空间，采用常规压缩存储后仍然具有随机存取特性。

(7) 稀疏矩阵是指非零元素个数相对于元素总数十分少的一类矩阵，其非零元素的分布没有规律性。

(8) 稀疏矩阵的压缩存储方式主要有三元组和十字链表表示，前者属顺序存储结构，后者属链式存储结构。

(9) 稀疏矩阵无论采用三元组还是十字链表存储方式后不再具有随机存取特性。

(10) 广义表是线性表的推广，其中的元素可以是原子，也可以是子广义表。

(11) 广义表中的数据元素是有相对次序的，其长度为最外层包含元素个数，其深度为所含括弧的重数。

(12) 一个广义表 $G=(a_1,a_2,\cdots,a_n)$ 可以拆分为表头和表尾，表头 $\mathrm{head}(G)=a_1$，表尾 $\mathrm{tail}(G)=(a_2,\cdots,a_n)$，表尾总是一个广义表。空广义表无表头和表尾。

(13) 广义表总是采用链式存储结构，该存储结构具有递归性，所以广义表算法很多都是递归算法。

(14) 广义表的两种递归算法设计方法是等价的，本质上是遍历所有结点，在实际中可以灵活运用。

6.2 教材中的练习题及参考答案

1. 如何理解数组是线性表的推广。

答：数组可以看成是线性表在下述含义上的扩展，即线性表中的数据元素本身也是一个线性表。在 $d(d \geq 1)$ 维数组中的每个数据元素都受着 d 个关系的约束，在每个关系中数据元素都有一个后继元素（除最后一个元素外）和一个前驱元素（除第一个元素外）。

因此，这 d 个关系中的任一关系就其单个关系而言仍是线性关系。例如，$m \times n$ 的二维数组的形式化定义如下：

$A=(D,R)$

其中：

$D = \{ a_{ij} \mid 0 \leq i \leq m-1, 0 \leq j \leq n-1 \}$　　//数据元素的集合

$R = \{ \mathrm{ROW}, \mathrm{COL} \}$

$\mathrm{ROW} = \{ <a_{i,j}, a_{i+1,j}> \mid 0 \leq i \leq m-2, 0 \leq j \leq n-1 \}$　　//行关系

$\mathrm{COL} = \{ <a_{i,j}, a_{i,j+1}> \mid 0 \leq i \leq m-1, 0 \leq j \leq n-2 \}$　　//列关系

2. 有三维数组 $a[0..7,0..8,0..9]$ 采用按行序优先存储，数组的起始地址是 1000，每个元素占用两个字节，试给出下面结果：

(1) 元素 $a_{1,6,8}$ 的起始地址。

(2) 数组 a 所占用的存储空间。

答：(1) $LOC(a_{1,6,8})=LOC(a_{0,0,0})+[1\times9\times10+6\times10+8]\times2=1000+316=1316$。

(2) 数组 a 所占用存储空间 $=8\times9\times10\times2=1440$ 字节。

3. 如果某个一维数组 A 的元素个数 n 很大，存在大量重复的元素，且所有元素值相同的元素紧挨在一起，请设计一种压缩存储方式使得存储空间更节省。

答：设数组的元素类型为 ElemType，采用一种结构体数组 B 来实现压缩存储，该结构体数组的元素类型如下。

```
struct
{   ElemType data;          //元素值
    int length;             //重复元素的个数
}
```

如数组 A[]={1,1,1,5,5,5,5,3,3,3,3,4,4,4,4,4,4}，共有 17 个元素，对应的压缩存储 B 为{1,3}，{5,4}，{3,4}，{4,6}}。从中看出，如果重复元素越多，采用这种压缩存储方式越节省存储空间。

4. 一个 n 阶对称矩阵 **A** 采用压缩存储在一维数组 B 中，则 B 中包含多少个元素?

答：通常 B 中包含 n 阶对称矩阵 **A** 的下三角和主对角线上的元素，其元素个数为 $1+2+\cdots+n=\frac{n(n+1)}{2}$，所以 B 包含 $\frac{n(n+1)}{2}$ 个元素。

5. 设 $n\times n$ 的上三角矩阵 **A**[0..$n-1$,0..$n-1$]已压缩到一维数组 B[0..m]中，若按列为主序存储，则 A[i][j]对应的 B 中存储位置 k 为多少? 给出推导过程。

答：对于上三角部分或者主对角中的元素 $A[i][j](i\leqslant j)$，在按列为主序存储时，前面有 $0\sim j-1$ 共 j 列，第 0 列有一个元素，第 1 列有两个元素，…，第 $j-1$ 列有 j 个元素，所以这 j 列的元素个数 $=1+2+\cdots+j=j(j+1)/2$；在第 j 列中，$A[i][j]$元素前有 $A[0..i-1,j]$共 i 个元素。所以 $A[i][j]$元素前有 $j(j+1)/2+i$ 个元素，而 B 的下标从 0 开始，所以 $A[i][j]$在 B 中的位置 $k=j(j+1)/2+i$。

6. 利用三元组存储任意稀疏数组 A，假设其中一个元素和一个整数占用的存储空间相同，问在什么条件下才能节省存储空间?

答：设稀疏矩阵 **A** 有 t 个非零元素，加上行数 rows、列数 cols 和非零元素个数 nums（也算一个三元组），那么三元组顺序表的存储空间总数为 $3(t+1)$，若用二维数组存储时占用的存储空间总数为 $m\times n$，只有当 $3(t+1)<m\times n$ 即 $t<m\times n/3-1$ 时，采用三元组存储才能节省存储空间。

7. 用十字链表存储一个有 k 个非 0 元素的 $m\times n$ 的稀疏矩阵，则其总的结点数为多少?

答：该十字链表有一个十字链表表头结点，$MAX(m,n)$个行、列表头结点。另外，每个非 0 元素对应一个结点，即 k 个元素结点。所以共有 $MAX(m,n)+k+1$ 个结点。

8. 求下列广义表运算的结果：

(1) head[(x,y,z)]

(2) tail[((a,b),(x,y))]

注意：为了清楚，在括号层次较多时将 head 和 tail 的参数用中括号表示。例如 head[G]、tail[G]分别表示求广义表 G 的表头和表尾。

答：(1) head[(x,y,z)]=x。

(2) tail[((a,b),(x,y))]=((x,y))。

9. 设定二维整数数组 $B[0..m-1,0..n-1]$的数据在行、列方向上都按从小到大的顺序排序，且整型变量 x 中的数据在 B 中存在。设计一个算法，找出一对满足 $B[i][j]=x$ 的 i、j 值，要求比较次数不超过 $m+n$。

解：从二维数组 B 的右上角的元素开始比较。每次比较有 3 种可能的结果：若相等，则比较结束；若 x 大于右上角的元素，则可断定二维数组的最上面一行肯定没有与 x 相等的数据，下次比较时搜索范围可减少一行；若 x 小于右上角的元素，则可断定二维数组的最右面一列肯定不包含与 x 相等的数据，下次比较时可把最右一列剔除出搜索范围。这样，每次比较可使搜索范围减少一行或一列，最多经过 $m+n$ 次比较就可找到要求的与 x 相等的元素。对应的程序如下：

```
#include <stdio.h>
#define M 3              //行数常量
#define N 4              //列数常量
void Find(int B[M][N],int x,int &i,int &j)
{   i = 0;j = N - 1;
    while (B[i][j]!= x)
        if(B[i][j]<x) i++;
        else j--;
}
int main()
{   int i,j,x = 11;
    int B[M][N] = {{1,2,3,4},{5,6,7,8},{9,10,11,12}};
    Find(B,x,i,j);
    printf("B[%d][%d] = %d\n",i,j,x);
    return 1;
}
```

10. 设计一个算法，计算一个用三元组表表示的稀疏矩阵的对角线元素之和。

解：对于稀疏矩阵三元组表 a，从 a.data[0]开始查看，若其行号等于列号，表示是一个对角线上的元素，则进行累加，最后返回累加值。算法如下：

```
bool diagonal(TSMatrix a,ElemType &sum)
{   sum = 0;
    if (a.rows!= a.cols)                  //行号不等于列号,返回 false
    {   printf("不是对角矩阵\n");
        return false;
    }
    for (int i = 0;i < a.nums;i++)
```

```
        if (a.data[i].r == a.data[i].c)        //行号等于列号
            sum += a.data[i].d;
    return true;
}
```

11. 设计一个算法 Same($g1$,$g2$),判断两个广义表 $g1$ 和 $g2$ 是否相同。

解:判断广义表是否相同的过程是若 $g1$ 和 $g2$ 均为 NULL,则返回 true;若 $g1$ 和 $g2$ 中一个为 NULL,另一不为 NULL,则返回 false;若 $g1$ 和 $g2$ 均不为 NULL,若同为原子且原子值不相等,则返回 false,若同为原子且原子值相等,则返回 Same($g1$->link,$g2$->link),若同为子表,则返回 Same($g1$->val.sublist,$g2$->val.sublist) & Same($g1$->link,$g2$->link)的结果,若一个为原子另一个为子表,则返回 false。对应的算法如下:

```
bool Same(GLNode *g1,GLNode *g2)
{   if (g1 == NULL && g2 == NULL)                   //均为 NULL 的情况
        return true;                                //返回真
    else if (g1 == NULL || g2 == NULL)              //一个为 NULL,另一不为 NULL 的情况
        return false;                               //返回假
    else                                            //均不空的情况
    {   if (g1->tag == 0 && g2->tag == 0)           //均为原子的情况
        {   if (g1->val.data!= g2->val.data)        //原子不相等
                return false;                       //返回假
            return(Same(g1->link,g2->link));        //返回兄弟比较的结果
        }
        else if (g1->tag == 1 && g2->tag == 1)      //均为子表的情况
            return(Same(g1->val.sublist,g2->val.sublist)
                & Same(g1->link,g2->link));
        else                                        //一个为原子,另一为子表的情况
            return false;                           //返回假
    }
}
```

6.3 补充练习题及参考答案



6.3.1 单项选择题

1. 数组 $\boldsymbol{a}$[0..5,0..6]的每个元素占 5 个单元,将其按列优先次序存储在起始地址为 1000 的连续内存单元中,则元素 a[5][5]的地址为_________。

A. 1175　　B. 1180　　C. 1205　　D. 1210

答:a[5][5]的地址=1000+(5×6+5)×5=1175。本题的答案是 A。

2. 设二维数组 $\boldsymbol{a}$[1..5,1..8],若按列优先的顺序存放数组的元素,则 a[4][6]元素的前面有_________个元素。

A. 6　　B. 28　　C. 29　　D. 40

答：当按列优先的顺序存储时，$a[4][6]$元素的前面有1～5列，每列5个元素，计25个元素，在第6列中，$a[4][6]$元素的前面有$a[1..3][6]$共3个元素，所以$a[4][6]$元素的前面有25＋3＝28个元素。本题的答案是B。

3. 矩阵$\boldsymbol{a}[m][n]$和矩阵$\boldsymbol{b}[n][p]$相乘，其时间复杂度为________。

A. $O(n)$　　B. $O(m\times n)$　　C. $O(m\times n\times p)$　　D. $O(n\times n\times n)$

答：C。

4. 对矩阵压缩存储是为了________。

A. 方便运算　　B. 节省存储空间

C. 方便存储　　D. 提高运算速度

答：B。

5. 一个n阶对称矩阵$\boldsymbol{a}[1..n,1..n]$采用压缩存储方式，将其下三角和主对角部分按行优先存储到一维数组$b[1..m]$中，则$a[i][j](i\geqslant j)$元素在b中的位置k是________。

A. $j(j-1)/2+i$　　B. $j(j-1)/2+i-1$

C. $i(i-1)/2+j$　　D. $i(i-1)/2+j-1$

答：对于下三角和主对角部分元素$a[i][j](i\geqslant j)$，前面第1行～第$i-1$行的元素个数分别为$1,2,\cdots,i-1$，计$i(i-1)/2$个元素，在第i行中，元素$a[i][j]$的前面有$a[i,1..j-1]$计$j-1$个元素，由于b的下标从1开始，所以该元素在b中的存储位置$k=i(i-1)/2+j-1+1=i(i-1)/2+j$。本题的答案是C。

6. 一个n阶对称矩阵$\boldsymbol{a}[1..n,1..n]$采用压缩存储方式，将其下三角和主对角部分按行优先存储到一维数组$b[1..m]$中，则$a[i][j](i<j)$元素在b中的位置k是________。

A. $j(j-1)/2+i$　　B. $j(j-1)/2+i-1$

C. $i(i-1)/2+j$　　D. $i(i-1)/2+j-1$

答：$a[i][j](i<j)$属于上三角部分的元素，它没有直接存储，但有$a[j][i]=a[i][j]$，而$a[j][i]$作为下三角和主对角部分的元素，由上例可以求出其存储位置为$k=j(j-1)/2+i$。本题的答案是A。

7. 一个n阶上三角矩阵$\boldsymbol{a}$按行优先顺序压缩存放在一维数组b中，则b中的元素个数是________。

A. n　　B. n^2　　C. $n(n+1)/2$　　D. $n(n+1)/2+1$

答：D。需要存储上三角和主对角部分的$n(n+1)/2$个元素，另外一个上三角的常量c。

8. 若将n阶下三角矩阵$\boldsymbol{a}$按列优先顺序压缩存放在一维数组$b[1..m]$中，$a_{1,1}$存于b_1中，则应存放到b_k中的非零元素$a_{i,j}(1\leqslant i\leqslant n,1\leqslant j\leqslant i)$的下标$i$、$j$与$k$的对应关系是________。

A. $\dfrac{j(2n-j+2)}{2}+i-j$　　B. $\dfrac{(j-1)(2n-j+2)}{2}+i-j+1$

C. $\dfrac{i(2n-i+2)}{2}+j-i+1$　　D. $\dfrac{i(2n-i+2)}{2}+j-i+1$

答：对于元素$a_{i,j}$，前有$j-1$列，第1列～第$j-1$列的元素个数分别为$n\sim n-j+2$，共计$\dfrac{(j-1)(2n-j+2)}{2}$个元素，而第j列上$a_{i,j}$之前有$i-j$个元素，也就是说，按这样的压缩

存放方式，$a_{i,j}$之前共有$\frac{(j-1)(2n-j+2)}{2}+i-j$个元素，因为$b$数组的下标从1开始，故$k=\frac{(j-1)(2n-j+2)}{2}+i-j+1$。本题的答案为B。

9. 对稀疏矩阵采用压缩存储，其缺点之一是________。

A. 无法判断矩阵有多少行、多少列

B. 无法根据行、列号查找某个矩阵元素

C. 无法根据行、列号直接计算矩阵元素的存储地址

D. 使矩阵元素之间的逻辑关系更加复杂

答：稀疏矩阵采用二维数组存储时具有随机存取特性，而采用压缩存储后不再具有随机存取特性。本题的答案为C。

10. 与三元组顺序表相比，稀疏矩阵用十字链表表示，其优点在于________。

A. 便于实现增加或减少矩阵中非零元素的操作

B. 便于实现增加或减少矩阵元素的操作

C. 可以节省存储空间

D. 可以更快地查找到某个非零元素

答：稀疏矩阵的三元组表示属顺序存储结构，其十字链表表示属链式存储结构，后者便于结点的增、删操作。本题的答案为A。

11. 在下列4个广义表中，长度为1、深度为4的广义表是________。

A. ((),((a)))　　　　B. ((((a),b)),c)

C. (((a,b),(c)))　　　　D. (((a,(b),c)))

答：D。

12. 空的广义表是指广义表________。

A. 深度为0　　　　B. 尚未赋值

C. 不含任何原子　　　　D. 不含任何元素

答：D。

13. 对于广义表((a,b),(()),(a,(b)))来说，其________。

A. 长度为4　　　　B. 深度为4

C. 有两个原子　　　　D. 有3个元素

答：D。该广义表的3个元素分别是(a,b)、(())和(a,(b))，它们都是子表。

14. 在广义表((a,b),c,((d),e),(f,j,(g),(h)))中，第4个元素的第3个元素是________。

A. 原子g　　　　B. 子表(g)

C. 原子e　　　　D. 子表((d),e)

答：第4个元素是(f,j,(g),(h))，它的第3个元素是子表(g)。本题的答案为B。

15. 已知广义表L=((x,y,z),(u,t,w))，从L表中取出原子t的运算是________。

A. head[tail[tail[L]]]　　　　B. tail[head[head[tail[L]]]]

C. head[tail[head[tail[L]]]]　　　　D. head[head[tail[tail[L]]]]

答：head[tail[tail[L]]]=head[tail[((u,t,w))]]=head[()]，运算出错(空表无表头)。

tail[head[head[tail[L]]]]=tail[head[head[((u,t,w))]]]

=tail[head[(u,t,w)]=tail[u],运算出错(原子不能求表尾)。

head[tail[head[tail[L]]]]=head[tail[head[((u,t,w)))]]

=head[tail[(u,t,w)]]=head[(t,w)]=t。

head[head[tail[tail[L]]]]=head[head[tail[((u,t,w))]]]

=head[head[()]],运算出错(空表无表头)。

本题的答案为C。

16. 广义表 A=(a,b,(c,d),(e,(f,g))),则 head[tail[head[tail[tail[A]]]]]的值为________。

A. (g) B. (d) C. (c) D. d

答:head[tail[head[tail[tail[A]]]]]=head[tail[head[tail[(b,(c,d),(e,(f,g)))]]]]

=head[tail[head[((c,d),(e,(f,g)))]]]=head[tail[(c,d)]]

=head[(d)]=d。

本题的答案为D。

6.3.2 填空题

1. 三维数组 a[0..4][0..6][0..8]共含有________个元素。

答:315。元素个数=5×7×9=315。

2. 一维数组 a 采用顺序存储方式,下标从0开始,每个元素占4个存储单元,a[8]的起始地址为100,则 a[11]的起始地址为________。

答:112。LOC(a[8])=LOC(a[0])+8×4=100,求得 a[0]元素的起始地址为68,LOC(a[11])=LOC(a[0])+11×4=112。

3. 设数组 a[1..60, 6..70]的基地址为2048,每个元素占两个存储单元,若以行序为主序顺序存储,则元素 a[32,58]的存储地址为________。

答:6102。在按行优先存储时,a[32,58]前面有1~31行,每行65个元素,计31×65=2015个元素,第32行前有 a[32,6..57]的元素,计12个元素。总计2015+12=2027,所以LOC(a[32,58])=2048+2027×2=6102。

4. 数组 a[1..10,-2..6,2..8]以行优先顺序存储,设第一个元素的首地址为100,每个元素占3个存储单元,则元素 a[5][0][7]的存储地址为________。

答:913。

LOC(a[5][0][7])=100+[(5-1)×(6-(-2)+1)×(8-2+1)+(0-(-2))×(8-2+1)+(7-2)]×3=913。

5. 一个 n 阶方阵,对于上三角部分(含主对角线)的元素 $a_{i,j}$,i、j 的关系是__①__;对于下三角部分(含主对角线)的元素 $a_{i,j}$,则 i、j 的关系是__②__。

答:① $i \leqslant j$ ② $i \geqslant j$。

6. 一个10阶对称矩阵 $\boldsymbol{a}$,采用以行序为主序只存储下三角和主对角部分的元素,每个元素占一个存储单元,且 a[0][0]的地址为1,则 a[8][5]的地址是________。

答:42。a[8][5]前有0~7行计36(1+2+…+8)个元素,第8行中 a[8][5]前有元素a[8,0..4]计5个元素,所以 a[8][5]前有36+5=41个元素,其压缩存储地址=1+41×

1=42。

7. 一个10阶对称矩阵 $\boldsymbol{a}$,采用以列序为主序只存储下三角部分的元素,每个元素占一个存储单元,且 $a[0][0]$ 的地址为1,则 $a[8][5]$ 的地址是________。

答:44。$a[8][5]$ 前有0~4列计40(10+9+8+7+6)个元素,第8行中 $a[8][5]$ 前有元素 $a[5..7,5]$ 计3个元素,所以 $a[8][5]$ 前有40+3=43个元素,其压缩存储地址=1+43×1=44。

8. 将一个 n 阶下三角矩阵采用压缩存储,共存储________个元素。

答:$n(n+1)/2+1$。

9. 广义表(((a,b,(),c),d),e,((f),g))的表头是 ① ,表尾是 ② 。

答:① ((a,b,(),c),d) ② (e,((f),g))。

10. 广义表(((a,b,(),c),d),e,((f),g))的长度是 ① ,深度是 ② 。

答:① 3 ② 4。

6.3.3 判断题

1. 判断以下叙述的正确性。

(1) 数组只能采用顺序存储结构。

(2) 特殊矩阵是指用途特殊的矩阵。

(3) 对称矩阵的行数和列数总是相同的。

(4) 设对称矩阵 $\boldsymbol{a}$ 按行优先将下三角和主对角部分压缩存储在一维数组 b 中,其中矩阵的第一个元素 a_{11} 存储在 $b[0]$,则元素 a_{ij} 在 b 中的存放位置 $k=i(i+1)/2+j$。

(5) 设 n 阶下三角矩阵 $\boldsymbol{a}$ 按行优先存储到一维数组 b 中,$a[0][0]$ 存放在 $b[0]$ 中,则 $a[i][i]$ 存放在 $b[i(i+1)/2]$ 中。

(6) 对角矩阵的特点是非零元素只出现在矩阵的两条对角线上。

(7) 在 $n(n>3)$ 阶三对角矩阵中,每一行都有3个非零的元素。

(8) 稀疏矩阵的特点是矩阵中的元素个数较少。

(9) 在稀疏矩阵的三元组存储结构中,每个元素仅包含非零元素的元素值。

(10) 稀疏矩阵采用三元组存储时具有随机存取特性,而采用十字链表存储时不具有随机存取特性。

答:(1) 错误。数组最适合采用顺序存储结构,但并不是说只能采用顺序存储结构。

(2) 错误。

(3) 正确。

(4) 错误。$k=i(i-1)/2+j-1$。

(5) 错误。元素 $a[i][i]$ 之前的元素个数 $=(1+2+\cdots+i)+i=i(i+1)/2+i$,所以 $a[i][i]$ 存放在 $b[i(i+1)/2+i]$ 中。

(6) 错误。

(7) 错误。

(8) 错误。

(9) 错误。三元组中每个非零元素包含行号、列号和元素值。

(10) 错误。稀疏矩阵的这两种压缩存储结构都不具有随机存取特性。

2. 判断以下叙述的正确性。

(1) 广义表的长度与广义表中含有多少个原子元素有关。

(2) 一个非空广义表的表尾总是一个广义表。

(3) 空的广义表是指广义表中不包含原子元素。

(4) 广义表的长度不小于其中任何一个子表的长度。

答：(1) 错误。

(2) 正确。

(3) 错误。

(4) 错误。例如,((a,b,c))的长度为1,而其子表(a,b,c)的长度为3。

6.3.4 简答题

1. 数组 $a[-3..5,2..5,-8..2]$有多少个元素?

答：该数组的第1维下标−3～5,长度为5−(−3)+1=9;第2维下标2～5,长度为5−2+1=4;第3维下标−8～2,长度为2−(−8)+1=11。所以元素个数=9×4×11=396个。

2. 为什么数组极少使用链式结构存储?

答：因为数组使用链式结构存储时不仅需要额外占用更多的存储空间,而且不再具有随机存取特性,使得相关操作更复杂。

3. 二维数组 $a[-3..3,-2..5]$的元素起始地址为1000,元素的长度为4,问按行优先存放和按列优先存放时 LOC($a[2][3]$)分别是多少?

答：在按行优先存放时,$a[2][3]$前面有第−3行～第1行,共1−(−3)+1=5行,每行有5−(−2)+1=8个元素,计40个元素;在第2行中,$a[2][3]$前面有元素 $a[2][-2..2]$,共2−(−2)+1=5个元素,所以 $a[2][3]$前面的元素个数=40+5=45,则 LOC($a[2][3]$)=1000+45×4=1180。

在按列优先存放时,$a[2][3]$前面有第−2行～第2列,共2−(−2)+1=5列,每列有3−(−3)+1=7个元素,计35个元素;在第3列中,$a[2][3]$前面有元素 $a[-3..1][3]$,共1−(−3)+1=5个元素,所以 $a[2][3]$前面的元素个数=35+5=40,则 LOC($a[2][3]$)=1000+35×4=1140。

4. 设二维数组 $a[0..9,0..19]$采用顺序存储方式,每个数组元素占用一个存储单元,$a[0][0]$的存储地址为200,$a[6][2]$的存储地址是322,问该数组采用的是按行优先存放还是按列优先存放?

答：这里有 $m=10$,$n=20$,$k=1$,一个 m 行 n 列的二维数组的顺序存储方式只能按行优先或列优先存放。

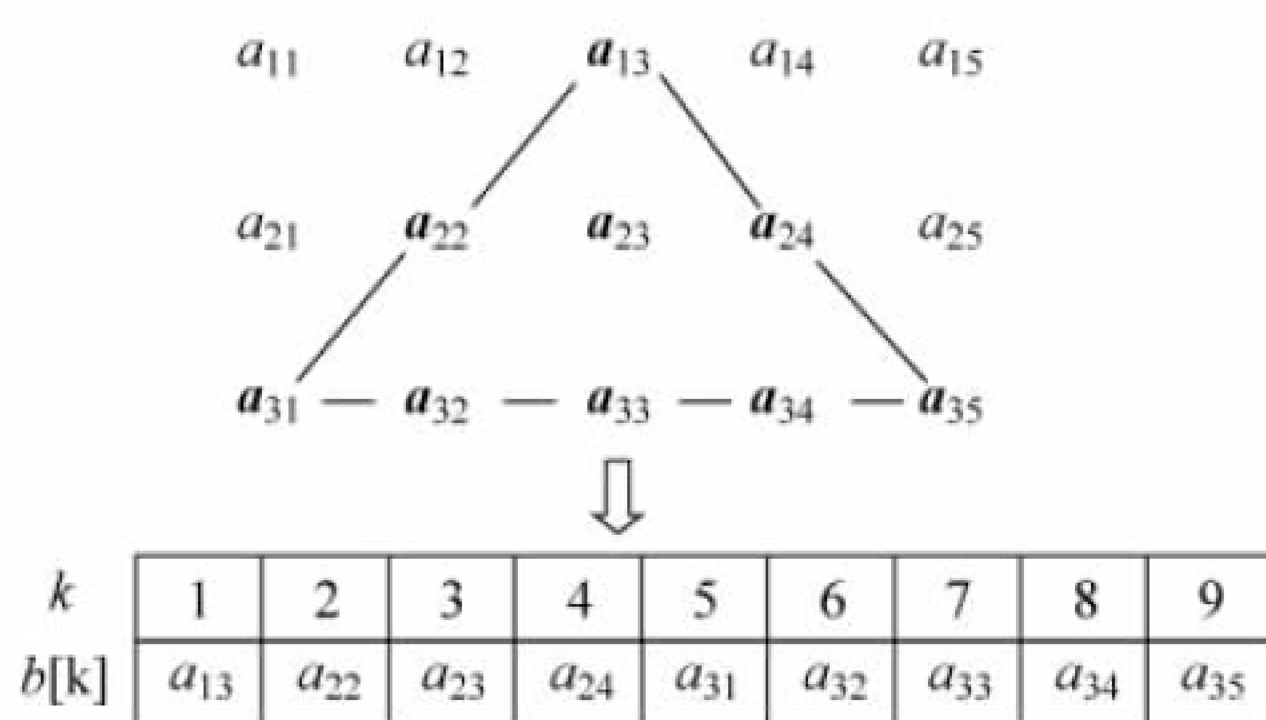
假设按行优先存放,有 $\mathrm{LOC}(a_{i,j})=\mathrm{LOC}(a_{0,0})+(i\times n+j)\times k$,对于 $a[6][2]$元素,其地址 LOC($a[6][2]$)=LOC($a[0][0]$)+[6×20+2]×1=322。

假设按列优先存放,有 $\mathrm{LOC}(a_{i,j})=\mathrm{LOC}(a_{0,0})+(j\times m+i)\times k$,对于 $a[6][2]$元素,其地址 LOC($a[6][2]$)=LOC($a[0][0]$)+[2×10+6]×1=226。

所以该数组采用的是按行优先存放方式。

5. 对于给定的数组 $a[1..n,1..2n-1]$,现将3个顶点分别为 $a[1][n]$、$a[n][1]$和

$a[n][2n-1]$的三角形上的所有元素按行序依次存放在一维数组 $b[1..n\times n]$中，图 6.2 所示为 $n=3$ 的情况。



k	1	2	3	4	5	6	7	8	9
b[k]	a_{13}	a_{22}	a_{23}	a_{24}	a_{31}	a_{32}	a_{33}	a_{34}	a_{35}

图 6.2　矩阵中三角形元素的存储方式

如果位于三角形上的元素 $a[i][j]$存放于 $b[k]$中，请给出计算公式 $k=f(n,i,j)$。对于该例中的 $a[3][4]$，根据计算公式可求得 $k=4$。

答：在三角形中，第 i 行中有 $2i-1$ 个元素，对于元素 $a[i][j]$，前面的 $1\sim i-1$ 行的元素个数之和$=1+3+\cdots+(2i-3)=(1+2i-3)(i-1)/2=(i-1)^2$。

第 1 行元素的列下标：n

第 2 行元素的列下标：$n-1,n,n+1$

第 3 行元素的列下标：$n-2,n-1,n,n+1,n+2$

…

第 i 行元素的列下标：$n-(i-1),\cdots$

归纳起来，第 i 行的第一个元素的下标为 $n-(i-1)$，所以在第 i 行中元素 $a[i][j]$的前面有 $j-[n-(i-1)]$个元素。

而 b 数组下标从 1 开始，所以 $k=(i-1)^2+j-[n-(i-1)]+1=(i-1)^2+i+j-n$。

本例中，$n=3$，对于元素 $a[3][4]$，$i=3$，$j=4$，$k=(i-1)^2+i+j-n=8$。

6. 阅读以下算法，指出其功能。若 $a[0..7]=\{1,2,3,4,5,6,7,8\}$，执行 fun($a$,8)后数组 a 的结果是什么？

```
void fun(int a[],int n)
{   int i = 0,j = 0;
    int tmp;
    while (j < n)
    {   if(a[j] % 2 == 1)                //a[j]为奇数
        {   tmp = a[i];                  //a[i]与 a[j]交换
            a[i] = a[j];
            a[j] = tmp;
            i++;
        }
        j++;
    }
}
```

答：在该算法中，用 i 表示当前奇数元素的个数，j 扫描所有元素，当 $a[j]$为奇数时，通

过交换将 $a[j]$ 插入到 $a[i]$ 处。所以算法的功能将含有 n 个整数的数组 a 中的所有奇数元素移到偶数元素的前面。

若 $a[0..7]=\{1,2,3,4,5,6,7,8\}$，执行 fun($a$,8)后数组 a 的结果是 $a[0..7]=\{1,3,5,7,2,6,4,8\}$。

7. 有一个 6 行 6 列的对称矩阵 $\boldsymbol{a}$，主对角线上的元素均为 0，其中主对角线以上部分的元素已按列优先顺序压缩存放在一维数组 b 中。根据以下 b 的内容画出 $\boldsymbol{a}$ 矩阵。

k：	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$b[k]$：	2	5	0	3	4	0	0	1	4	2	6	3	0	1	2

答：对于 6×6 对称矩阵 $\boldsymbol{a}$，首先将主对角元素置为 0，根据 b 数组确定上三角部分元素，再由对称性确定下角部分的元素。a 的结果如下：

$$\begin{bmatrix} 0 & 2 & 2 & 0 & 0 & 2 \\ 2 & 0 & 5 & 3 & 0 & 6 \\ 2 & 5 & 0 & 4 & 1 & 3 \\ 0 & 3 & 4 & 0 & 4 & 0 \\ 0 & 0 & 1 & 4 & 0 & 1 \\ 2 & 6 & 3 & 0 & 1 & 0 \end{bmatrix}$$

8. 一个 $n\times n$ 的对称矩阵存入内存，在采用压缩存储和采用非压缩存储时占用的内存空间分别是多少？求压缩存储时的压缩比。

答：若采取压缩存储，其容量为 $n(n+1)/2$，若不采用压缩存储，其容量为 n^2。采用压缩存储时的压缩比 $=\dfrac{n(n+1)/2}{n^2}=\dfrac{1}{2}+\dfrac{1}{2n}$。

9. 对于特殊矩阵 $\boldsymbol{a}$(对称矩阵、上/下三角矩阵、三对角矩阵)，采用压缩方式存储到一维数组 b 中，$\boldsymbol{a}$ 中的元素 a_{ij} 存储在 b_k 中，给出计算 $k=f(i,j)$ 的一般过程。

答：计算 $k=f(i,j)$ 的一般过程如下。

(1) 根据压缩存储方式(按上三角还是按下三角存放、按行还是按列优先存放)，计算出元素 a_{ij} 前面 $1\sim i-1$ 行或者 $1\sim j-1$ 列有多少个元素(如 k_1 个)。

(2) 计算第 i 行或者第 j 列中元素 a_{ij} 前面有多少个元素(如 k_2 个)。

(3) 确定数组 b 的下标从 1 开始($k_3=1$)还是从 0 开始($k_3=1$)，或者其他。

(4) $k=k_1+k_2+k_3$。

(5) 若有常量需要存储，通常存放在最后位置。

10. 设 $n\times n$ 的上三角矩阵 $a[0..n-1,0..n-1]$ 已压缩存储到一维数组 $b[s..t]$ 中，若按列为主序存储，则 $a[i][j]$ 对应的 b 中存储位置 k 为多少，给出推导过程。

答：对于上三角部分的 $a[i][j](i\leqslant j)$ 元素，在按列为主序存储时，前面有 $0\sim j-1$ 共 j 列，第 0 列有一个元素，第 1 列有两个元素，…，第 $j-1$ 列有 j 个元素，这 j 列的元素个数 $=1+2+\cdots+j=j(j+1)/2$；在第 j 列中：$a[i][j]$ 元素前有 $a[0..i-1,j]$ 共 i 个元素，$a[i][j]$ 元素的前面共有 $j(j+1)/2+i$ 个元素。数组 b 的下标从 s 开始。所以，$k=j(j+1)/2+i+s$。

11. 已知广义表 $A=(((\mathrm{a})),(\mathrm{b}),\mathrm{c},(\mathrm{a}),(((\mathrm{d},\mathrm{e}))))$，画出它的存储结构图；给出表的

长度与深度；用求表头、表尾的方式求出 e。

答：广义表 A 的一种存储结构如图 6.3 所示。A 的长度为 5，深度为 4。

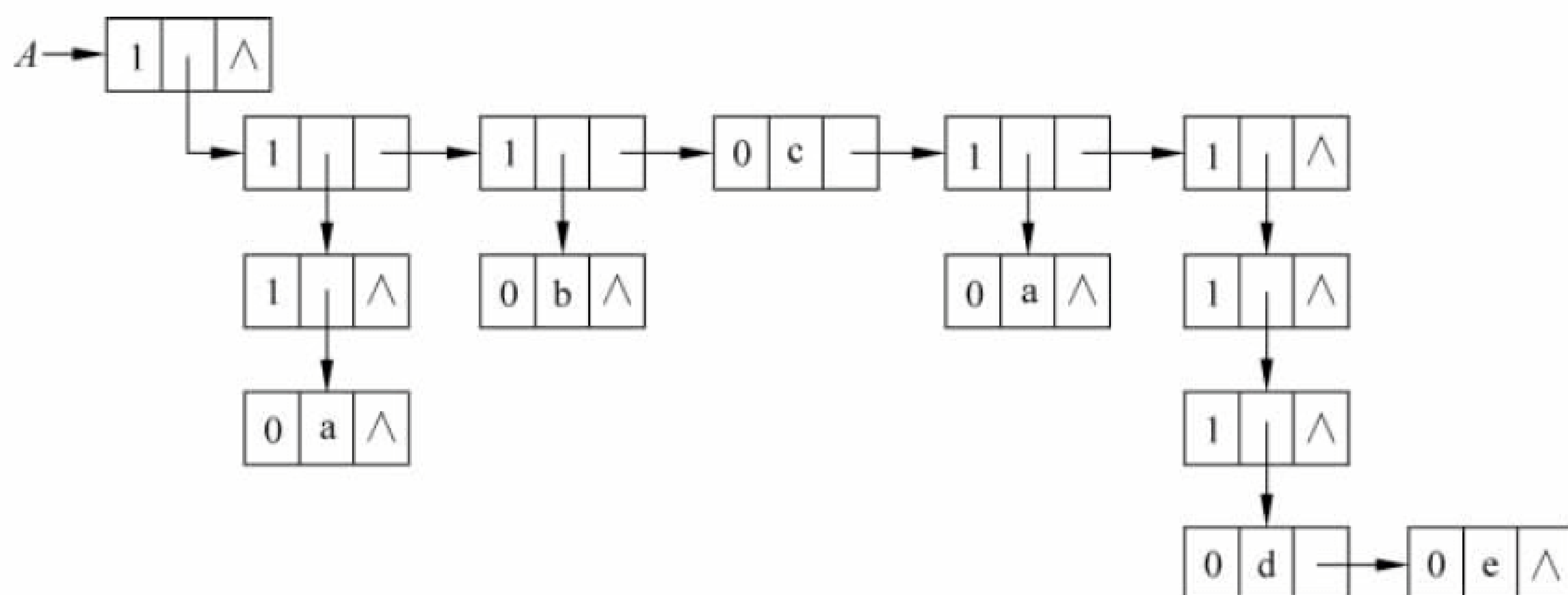


图 6.3　广义表 A 的存储结构

求出原子 e 的过程是对广义表 A 执行 4 次求表尾操作得到((((d,e)))),然后执行 3 次求表头操作得到(d,e),再执行一次求表尾操作得到(e),最后执行一次求表头操作得到 e，即方式是 head[tail[head[head[head[tail[tail[tail[tail[A]]]]]]]]]。

12. 已知广义表(a,(b,(a,b)),((a,b),(a,b))),试完成以下要求：

(1) 画出该广义表存储结构。

(2) 计算该广义表的表头和表尾。

(3) 计算该广义表的深度。

答：(1)该广义表的一种存储结构如图 6.4 所示。

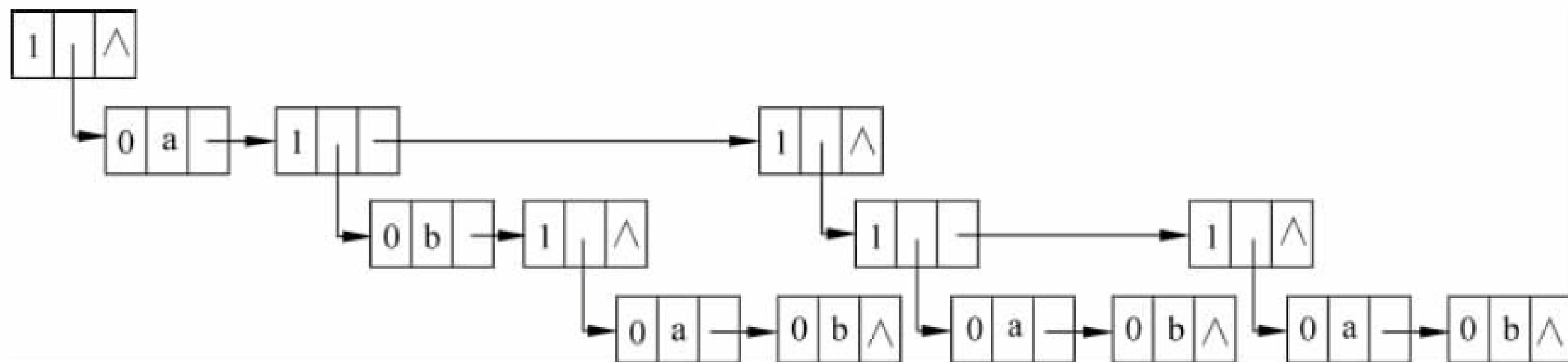


图 6.4　广义表的存储结构

(2) 该广义表的表头为 a，表尾为((b,(a,b)),((a,b),(a,b)))。

(3) 该广义表的深度为 3。

6.3.5　算法设计题

1.【数组算法】有一个含有 n 个整数元素数组 $a[0..n-1]$，设计一个算法求其中最后一个最小元素的下标。

解：设最后一个最小元素的下标为 mini，初值为 0。i 从 1 到 $n-1$ 循环，当 $a[i]<=a[\text{mini}]$时置 mini$=i$。对应的算法如下：

```
void FindMin(int a[],int n,int &mini)
{   int i;
```

```
    mini = 0;
    for (i = 1;i < n;i++)
        if (a[i]<= a[mini])
            mini = i;
}
```

2.【数组算法】设计一个算法，求一个 m 行 n 列的二维整型数组 a 的下三角部分的所有元素之和，当 $m \neq n$ 时返回 false，否则返回 true。

解：当 $m \neq n$ 时返回 false，否则置 s 为 0，下三角部分的元素为 $a[i][j]$（$1 \leqslant i < m, i > j$），用两重循环累加其和，最后返回 true。对应的算法如下：

```
bool LowDiag(int a[M][N],int m,int n,int &s)
{   int i,j;
    if (m!= n)
        return false;
    s = 0;
    for (i = 1;i < m;i++)
        for (j = 0;j < i;j++)
            s += a[i][j];
    return true;
}
```

3.【数组算法】假设有一个 m 行 n 列的二维数组 a，其中所有元素为整数。其大量的运算是求左上角为 $a[i,j]$、右下角为 $a[s,t]$（$i < s, j < t$）的子矩阵的所有元素之和。请设计一个时间复杂度为 $O(1)$ 的算法求给定子矩阵中的所有元素之和（本题是国外一家著名软件公司的面试题）。

解：建立一个 m 行 n 列的二维数组 b，$b[i,j]$ 为 a 中左上角为 $a[0,0]$、右下角为 $a[i,j]$ 的子矩阵的所有元素之和。由数组 a 求出数组 b 的算法如下：

```
void sum(int a[ ][MAXN],int b[ ][MAXN],int m,int n)
{   int i,j;
    b[0][0] = a[0][0];
    for (i = 1;i < m;i++)               //求 b 的第 0 列
        b[i][0] = b[i - 1][0] + a[i][0];
    for (j = 1;j < n;j++)               //求 b 的第 0 行
        b[0][j] = b[0][j - 1] + a[0][j];
    for (i = 1;i < m;i++)               //求 b[i][j]
        for (j = 1;j < n;j++)
            b[i][j] = a[i][j] + b[i - 1][j] + b[i][j - 1] - b[i - 1][j - 1];
}
```

该算法的时间复杂度为 $O(m \times n)$。在求出 b 数组后，求数组 a 中左上角为 $a[i,j]$、右下角为 $a[s,t]$（$i < s, j < t$）的子矩阵的所有元素之和就可以利用数组 b 来实现，如图 6.5 所示，其值为 $b[s][t] - b[s][j-1] - b[i-1][t] + b[i-1][j-1]$。对应的算法如下：

```
int submat(int b[ ][MAXN],int i,int j,int s,int t)
{   int sum;
```

```
        if (i == 0 && j == 0)
            return b[s][t];
        else
            sum = b[s][t] - b[s][j - 1] - b[i - 1][t] + b[i - 1][j - 1];
        return sum;
    }
```

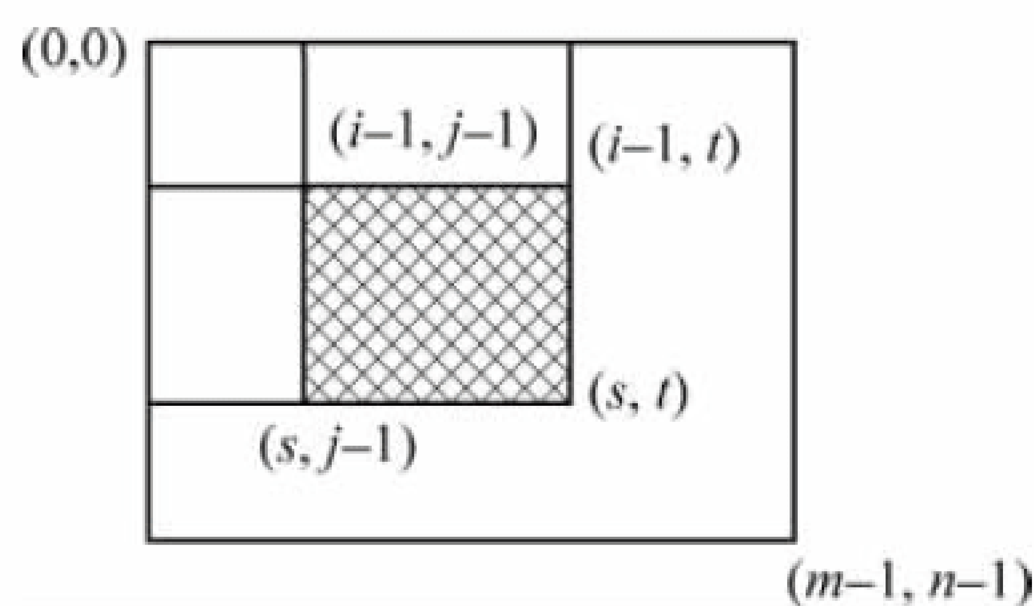


图 6.5 子矩阵的所有元素之和$=b[s][t]-b[s][j-1]-b[i-1][t]+b[i-1][j-1]$

显然，该算法的时间复杂度为 $O(1)$。尽管 sum 算法的时间复杂度为 $O(m\times n)$，但只需要执行一次，而 submat 算法需要大量应用，所以这种设计是十分经济的(这实际上就是数据结构课程的精髓)。

4.【数组算法】设有一个整型数组 a，设计一个算法，使 $a[i]$的值变为 $a[0]$到 $a[i-1]$中小于原 $a[i]$值的个数。分析该算法的时间复杂度。

解：i 从 $n-1$ 到 0 循环，累计 $a[0..i-1]$中大于 $a[i]$的元素个数 c，置 $a[i]$为 c。本算法的时间复杂度为 $O(n^2)$。对应的算法如下：

```
void fun(int a[ ],int n)
{   int i,j,c;
    for (i = n - 1;i >= 0;i -- )
    {   c = 0;
        for (j = 0;j < i;j++)
            if (a[j]<a[i]) c++;
        a[i] = c;
    }
}
```

5.【数组算法】设有一个元素递增的整型数组 a，且所有元素均不相同。设计一个高效的算法，使 $a[i]$的值变为 $a[0]$到 $a[i-1]$中小于原 $a[i]$值的个数。分析该算法的时间复杂度。

解：由于数组 a 中元素递增且所有元素均不相同，置 $a[0]=0$，用 i 从 0 到 $n-1$ 循环：tmp 保存 $a[i-1]$元素值，若 $a[i]>$tmp，置 $a[i]=a[i-1]+1$，否则 $a[i]=a[i-1]$。本算法的时间复杂度为 $O(n)$。对应的算法如下：

```
void fun(int a[ ],int n)
{   int i,tmp = a[0];
    a[0] = 0;
    for (i = 1;i < n;i++)
```

```
    {   if (a[i]> tmp)
        {   tmp = a[i]; a[i] = a[i-1]+1; }
        else
        {   tmp = a[i]; a[i] = a[i-1]; }
    }
}
```

6.【数组算法】给定一个有 $n(n\geqslant1)$个整数的序列用整型数组 a 存储，要求求出其中最大连续子序列的和。例如序列(−2,11,−4,13,−5,−2)的最大子序列和为20，序列(−6,2,4,−7,5,3,2,−1,6,−9,10,−2)的最大子序列和为16。说明算法的时间复杂度。

解：设含有 n 个整数的序列 $a[0..n-1]$的任何连续子序列 $a[i..j]$($i\leqslant j,0\leqslant i\leqslant n-1,i\leqslant j\leqslant n-1$)，求出它的所有元素之和 thisSum，并通过比较将最大值存放在 maxSum 中，最后返回 maxSum。

本算法穷举所有连续子序列(一个连续子序列由起始下标 i 和终止下标 j 确定)来求解，用了三重循环，所以有：

$$T(n)=\sum_{i=0}^{n-1}\sum_{j=i}^{n-1}\sum_{k=i}^{j}1=\sum_{i=0}^{n-1}\sum_{j=i}^{n-1}(j-i+1)=\frac{1}{2}\sum_{i=0}^{n-1}(n-i)(n-i+1)=O(n^3)$$

对应的算法如下：

```
long maxSubSum1(int a[],int n)
{   int i,j,k;
    long maxSum = a[0],thisSum;
    for (i = 0;i < n;i++)               //两重循环穷举所有的连续子序列
    {   for (j = i;j < n;j++)
        {   thisSum = 0;
            for (k = i;k <= j;k++)
                thisSum += a[k];
            if (thisSum > maxSum)       //通过比较求最大连续子序列之和
                maxSum = thisSum;
        }
    }
    return maxSum;
}
```

7.【数组算法】设计一个算法，将一维数组 $A[0..n\times n-1]$($n\leqslant10$)中的元素按蛇形方式存放在二维数组 $B[0..n-1,0..n-1]$中。即：

$B[0][0]=A[0]$

$B[0][1]=A[1],B[1][0]=A[2]$

$B[2][0]=A[3],B[1][1]=A[4],B[0][2]=A[5]$

…

依此类推，如图6.6所示。

$$\begin{bmatrix} A[0] & A[1] & A[5] & A[6] & \cdots \\ A[2] & A[4] & A[7] & A[13] & \cdots \\ A[3] & A[8] & A[12] & \cdots & \cdots \\ A[9] & A[11] & \cdots & \cdots & \cdots \\ A[10] & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots \end{bmatrix}$$

图 6.6　蛇形二维数组

解：以 $n=4$ 为例，生成的二维矩阵如图 6.7 所示，其中有 7 条斜线，每条斜线的元素个数为 gs，奇数斜线的元素编号从下向上递增，偶数斜线的元素编号从上向下递增。

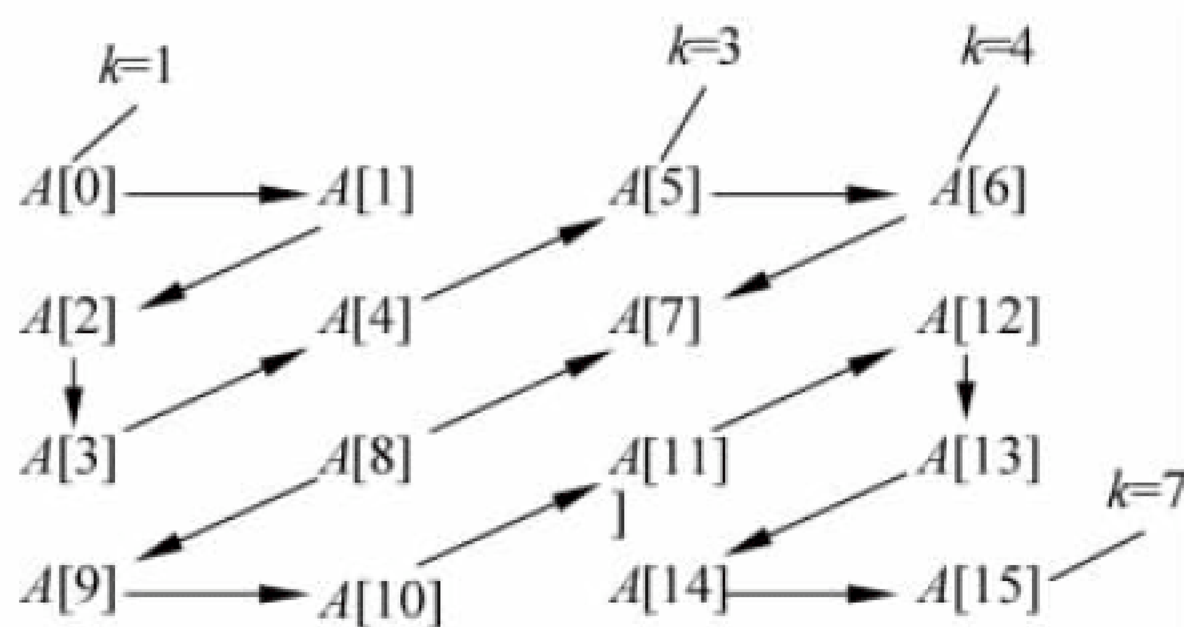


图 6.7　$n=4$ 的蛇形二维数组

本题的算法如下：

```
void func(int A[n * n],int B[n][n])
{   int i,j,k,m,g,gs;
    m = 0;
    for (k = 1;k <= 2 * n - 1;k++)          //对于每条对角线循环一次
    {   if (k < n)
            gs = k;                         //gs 为第 k 条斜线上的元素个数
        else
            gs = 2 * n - k;
        for (g = 1;g <= gs;g++)
        {   if (k % 2 == 1)                 //k 为奇数的情况,从下向上方递增
            {   i = gs - g;
                j = g - 1;
            }
            else                            //k 为偶数的情况,从上向下方递增
            {   i = g - 1;
                j = gs - g;
            }
            if (k > n)                      //考虑第 n + 1 到 2n - 1 的斜线
            {   i = i + n - gs;
                j = j + n - gs;
            }
            B[i][j] = A[m];m++;
        }
    }
}
```

8.【对称矩阵压缩存储算法】两个 n 阶整型对称矩阵 **A**、**B** 采用压缩存储方式，均按行优先顺序存放其下三角和主对角线的各元素。设计一个算法求 **A**、**B** 的乘积 C，要求 C 直接用二维数组表示。

解：对于两个 n 阶对称矩阵 **A**、**B**，在求乘积 C 数组时，$C[i][j]=\sum_{k=0}^{n-1}B[i][k]\times A[k][j]$。由于 **A**、**B** 均采用 a、b 压缩存储，设计 findk(int i,int j)算法由 i、j 求压缩存储中的下标。在乘积式中，求 $k1$=findk(i,k)，$k2$=findk(k,j)，$A[i][k]$用 $a[k1]$替代，$B[k][j]$用 $b[k2]$替代即可。本题的算法如下：

```
int findk(int i,int j)              //由 i、j 求压缩存储中的 k 下标
{   if (i >= j)
        return (i * (i + 1)/2 + j);
    else
        return (j * (j + 1)/2 + i);
}
void Mult(int a[],int b[],int c[M][N],int n)
{   int i,j,k,k1,k2;
    int s;
    for (i = 0;i < n;i++)
        for (j = 0;j < n;j++)
        {   s = 0;
            for (k = 0;k < n;k++)
            {   k1 = findk(i,k);
                k2 = findk(k,j);
                s += a[k1] * b[k2];
            }
            c[i][j] = s;
        }
}
```

9.【稀疏矩阵算法】稀疏矩阵只存放其非零元素的行号、列号和元素值，用一维数组顺序存放，行号−1 作为结束标志，试写出两个稀疏矩阵相加的算法。

解：设有一个稀疏矩阵如下。

$$\begin{bmatrix}1 & 0 & 0 & 2\\ 0 & 0 & 3 & 0\\ 0 & 4 & 0 & 5\end{bmatrix}$$

则对应的非零元素位置及值为(0,0,1)，(0,3,2)，(1,2,3)，(2,1,4)，(2,3,5)，所以一维数组 R 为 $R[0]=0$，$R[1]=0$，$R[2]=1$，$R[3]=0$，$R[4]=3$，$R[5]=2$，$R[6]=1$，$R[7]=2$，$R[8]=3$，$R[9]=2$，$R[10]=1$，$R[11]=4$，$R[12]=2$，$R[13]=3$，$R[14]=5$，$R[15]=-1$。

从而可以利用两个数组 A 和 B 来存放两个稀疏矩阵，数组 C 可用来存放两个稀疏矩阵相加的和。对应的算法如下：

```
void Add(int A[],int B[],int C[])
{   int i = 0,j = 0,k = 0,sum;
    while ((A[i]!= - 1 && B[j]!= - 1)
```

```
    {   if (A(i] == B[j])             //按列优先比较
        {   if (A[i+1] == B[j+1])  //比较列号
            {   sum = A[i+2] + B[j+2];
                if (sum!= 0)          //和不为零时
                {   C[k] = A[i];C[k+1] = A[i+1];C[k+2] = sum;
                    k = k+3;
                }
                i = i+3;
                j = j+3;
            }
            else if (A[i+1]< B[j+1])
            {   C[k] = A[i];C[k+1] = A[i+1];C[k+2] = A[i+2];
                k = k+3;i = i+3;
            }
            else                          //A[i+1]> B[j+1]
            {   C[k] = B[j];C[k+1] = B[j+1];C[k+2] = B[j+2];
                k = k+3;j = j+3;
            }
        }
        else if (A[i]< B[j])
        {   C[k] = A[i];C[k+1] = A[i+1];C[k+2] = A[i+2];
            k = k+3;i = i+3;
        }
        else                              //A[i]> B[j]
        {   C[k] = B[j];C[k+1] = B[j+1];C[k+2] = B[j+2];
            k = k+3;j = j+3;
        }
    }
    if (A[i] == -1 && B[j]!= -1)
    {   C[k] = B[j];C[k+1] = B[j+1];C[k+2] = B[j+2];
        k = k+3;j = j+3;
    }
    if (A[i]!= -1 && B[j] == -1)
    {   C[k] = A[i];C[k+1] = A[i+1];C[k+2] = A[i+2];
        k = k+3;i = i+3;
    }
    C[k] = -1;
}
```

10.【广义表算法】设计一个算法 change(g,s,t),将一个广义表 g 中的所有原子 s 替换成 t。例如 cgange("(a,(a,b),((a,b),c))",'a','x'),返回的结果为"(x,(x,b),((x,b),c))"。

解：广义表的替换过程的递归模型 $f(g,s,t)$ 如下。

$$f(g,s,t)\equiv\begin{cases}\text{不做任何事件} & \text{若 } g=\text{NULL}\\ \text{将 } g \text{ 原子值替换成 } t\text{；} f(g->link,s,t) & \text{若 } g \text{ 为原子结点且 } g->val.data=s\\ f(g->link,s,t) & \text{其他情况}\end{cases}$$

对应的算法如下：

```
void Change(GLNode *&g,ElemType s,ElemType t)
{   if (g!= NULL)
```

```
        {   if (g->tag==1)               //子表的情况
                Change(g->val.sublist,s,t);
            else if (g->val.data==s)     //原子且 data 域值为 s 的情况
                g->val.data=t;
            Change(g->link,s,t);
        }
    }
```

11.【广义表算法】假设广义表 g 中的原子为小写字母，设计一个算法 MaxAtom(g)，求出一个广义表 g 中最大的原子。例如，MaxAtom((a,(b),d,c)返回的结果为 d。

解：算法思路是对于广义表的每个元素进行循环，若为子表，递归在该子表中求最大的原子。对应的算法如下：

```
ElemType MaxAtom(GLNode *g)
{   char max='a',m;                         //max 赋初值为最小小写字母'a'
    while (g!=NULL)
    {   if (g->tag==1)                      //子表的情况
        {   m=MaxAtom(g->val.sublist);      //对子表递归调用
            if (m>max) max=m;
        }
        else
        {   if (g->val.data>max)            //为原子时进行原子比较
                max=g->val.data;
        }
        g=g->link;
    }
    return max;
}
```

12.【广义表算法】假设广义表 g 中的原子为整数值，设计一个算法 AtomSum(g)，计算一个广义表 g 中所有原子的和。

解：计算广义表的原子个数的递归模型 $f(g)$ 如下：

$$f(g)=\begin{cases}0 & \text{若 } g=\text{NULL}\\ g->val.data+f(g->link) & \text{若 } g \text{ 为原子}\\ f(g->val.sublist)+f(g->link) & \text{若 } g \text{ 为子表}\end{cases}$$

对应的算法如下：

```
int AtomSum(GLNode *g)
{   int s=0;
    if (g!=NULL)
    {   if (g->tag==1)               //为子表时
            s+=AtomSum(g->val.sublist);
        else                         //为原子时
            s+=g->val.data;
        s+=AtomSum(g->link);         //求兄弟的原子之和
    }
    return s;
}
```

第 7 章 树和二叉树

7.1 本章知识体系



1. 知识结构图

本章的知识结构如图7.1所示。



图7.1 第7章知识结构图

2. 基本知识点

(1) 树的递归特点和树的相关术语。
(2) 树的性质、树的遍历和树的存储结构。
(3) 二叉树与树/森林之间的转换方法。
(4) 二叉树的递归特点、二叉树的性质和二叉树的两种存储结构。
(5) 完全二叉树和满二叉树的特点。
(6) 二叉树的先序、中序和后序遍历递归和非递归算法设计。
(7) 二叉树的层次遍历算法设计。
(8) 二叉树遍历算法的应用。
(9) 二叉树的构造过程。

(10) 线索二叉树的特点、构造和遍历过程。

(11) 哈夫曼树的特点、哈夫曼树的构造过程和哈夫曼编码的生成。

(12) 灵活地运用二叉树这种数据结构解决一些综合应用问题。

3. 要点归纳

(1) 树用来表示具有层次结构的数据。

(2) 在一棵树中根结点没有前驱结点，其余每个结点都有唯一的前驱结点。

(3) 度为 m 的树至少有一个结点的度为 m，且没有度大于 m 的结点。例如，度为 3 的树至少有 4 个结点。

(4) 含有 n 个结点的 m 次树，$n=n_0+n_1+\cdots+n_m$，所有结点度之和 $=n_1+2n_2+\cdots+mn_m$。

(5) 对于含有 n 个结点的树(或者二叉树)，无论度为多少，其分支数或所有结点度之和均为 $n-1$。

(6) 对于高度为 h 的 m 次树，当为满 m 次树时结点个数最多。

(7) 树的遍历运算主要有先根遍历、后根遍历和层次遍历 3 种。

(8) 树的存储结构主要有双亲存储结构、孩子链存储结构和孩子兄弟链存储结构。

(9) 二叉树或者是一棵空树，或者是一棵由一个根结点和两棵互不相交的分别称为根结点的左子树和右子树所组成的非空树，左子树和右子树又同样都是一棵二叉树。

(10) 二叉树和树都属于树形结构，但二叉树并不是特殊的树。

(11) 度为 2 的树和二叉树是不同的。

(12) 二叉树中所有结点的度均小于或等于 2。结点个数 $=n_0+n_1+n_2$，所有结点度之和 $=n-1=n_1+2n_2$，所以 $n_0=n_2+1$。

(13) 在二叉树中，根结点的层次(或深度)为 1，一个结点的层次是其双亲结点的层次加 1。

(14) 在二叉树中，二叉树的高度 h 等于从根结点到叶子结点的最长路径上的结点个数。

(15) 满二叉树中所有分支结点的度皆为 2，即 $n_1=0$，且所有叶子结点在同一层上。

(16) 若满二叉树的结点数为 n，其树形是唯一确定的，高度 $h=\log_2(n+1)$。

(17) 完全二叉树中除最后一层以外，其余层都是满的，并且最后一层的右边缺少连续若干个结点。

(18) 完全二叉树中单分支结点个数 n_1 只能为 1 或 0，当结点总数 n 为奇数时，$n_1=0$；当 n 为偶数时，$n_1=0$。

(19) 对于结点个数为 n 的完全二叉树，其树形是唯一确定的。也就是说，可以由 n 计算出 n_0、n_1、n_2 和高度 h，$h=\lceil\log_2(n+1)\rceil$。

(20) 二叉树的存储结构主要有顺序存储结构和二叉链存储结构两种。

(21) 在含有 n 个结点的二叉链中，通过根结点指针来唯一标识该二叉链，其中空指针域个数为 $n+1$。

(22) 二叉树的遍历方式主要有先序遍历、中序遍历、后序遍历和层次遍历。

(23) 在二叉树的先序遍历、中序遍历和后序遍历中，所有左子树均在右子树之前遍历。这 3 种遍历算法可以采用递归实现，也可以用栈转换为非递归算法。

(24) 在非递归后序遍历中，当出栈并访问一个结点时，栈中恰好包含它的所有祖先。

(25) 在设计二叉树的递归算法时，通常以整棵二叉树的求解为“大问题”，而左、右子树的求解为“小问题”。假设左、右子树可以求解，推导出“大问题”的求解关系，从而得到递归体，再根据递推方向考虑一个特殊情况(如空树或只有一个结点的二叉树)给出递归出口，得到递归模型，在此基础上写出递归算法。

(26) 假设二叉树中的结点值均不相同，由先序序列和中序序列可以唯一确定一棵二叉树。

(27) 假设二叉树中的结点值均不相同，由后序序列和中序序列可以唯一确定一棵二叉树。

(28) 假设二叉树中的结点值均不相同，由层次遍历序列和中序序列可以唯一确定一棵二叉树。

(29) 在将一棵非空树转换成二叉树时，树的根结点变为二叉树的根结点。树的每个结点的最左孩子(长子)变为二叉树的左孩子，其他孩子变成该孩子的右下结点。

(30) 在将一个森林转换成二叉树时，整个森林转换成一棵二叉树。第 1 棵树的根结点变为二叉树的根结点，第 1 棵树的其他结点变为二叉树根结点左子树中的结点，森林的其他树中结点变为二叉树根结点的右子树中的结点。

(31) 线索二叉树是由二叉链存储结构变化而来的，将原来的空链域改为某种遍历次序下该结点的前驱结点和后继结点的指针。

(32) 中序线索二叉树可以采用不需要栈的非递归算法来实现中序遍历，对应的空间复杂度为 $O(1)$。

(33) 中序线索二叉树仅仅有利于中序遍历，并不能提高先序遍历和后序遍历的效率。

(34) 哈夫曼树是带权路径长度最小的二叉树。

(35) 哈夫曼树中权值较小的叶子结点一般离根结点较远。

(36) 哈夫曼树中的单分支结点个数为 0。在含有 m 个叶子结点的哈夫曼树中，其结点总数为 $2m-1$。

(37) 在一组字符的哈夫曼编码中，任何字符的编码不可能是另一个字符编码的前缀。

7.2 教材中的练习题及参考答案

1. 有一棵树的括号表示为 A(B,C(E,F(G)),D)，回答下面的问题：

(1) 指出树的根结点。

(2) 指出这棵树的所有叶子结点。

(3) 结点 C 的度是多少?

(4) 这棵树的度为多少?

(5) 这棵树的高度是多少?

(6) 结点 C 的孩子结点是哪些?

(7) 结点 C 的双亲结点是谁?

答:该树对应的树形表示如图 7.2 所示。

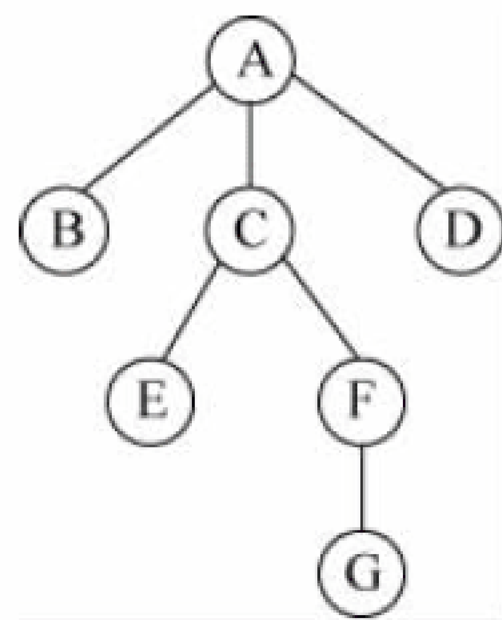


图 7.2　一棵树

(1) 这棵树的根结点是 A。

(2) 这棵树的叶子结点是 B、E、G、D。

(3) 结点 C 的度是 2。

(4) 这棵树的度为 3。

(5) 这棵树的高度是 4。

(6) 结点 C 的孩子结点是 E、F。

(7) 结点 C 的双亲结点是 A。

2. 若一棵度为 4 的树中度为 2、3、4 的结点个数分别为 3、2、2,则该树的叶子结点的个数是多少?

答:结点总数 $n=n_0+n_1+n_2+n_3+n_4$,又由于除根结点以外,每个结点都对应一个分支,所以总的分支数等于 $n-1$。而一个度为 $i(0\leqslant i\leqslant 4)$的结点的分支数为 i,所以有总分支数$=n-1=1\times n_1+2\times n_2+3\times n_3+4\times n_4$。综合两式得 $n_0=n_2+2n_3+3n_4+1=3+2\times 2+3\times 2=14$。

3. 为了实现以下各种功能,x 结点表示该结点的位置,给出树的最适合的存储结构:

(1) 求 x 和 y 结点的最近祖先结点。

(2) 求 x 结点的所有子孙结点。

(3) 求根结点到 x 结点的路径。

(4) 求 x 结点的所有右边兄弟结点。

(5) 判断 x 结点是否为叶子结点。

(6) 求 x 结点的所有孩子结点。

答:(1) 双亲存储结构。

(2) 孩子链存储结构。

(3) 双亲存储结构。

(4) 孩子兄弟链存储结构。

(5) 孩子链存储结构。

(6) 孩子链存储结构。

4. 设二叉树 bt 的一种存储结构如表 7.1 所示。其中,bt 为树根结点指针,lchild、rchild 分别为结点的左、右孩子指针域,在这里使用结点编号作为指针域值,0 表示指针域值为空;data 为结点的数据域。请完成下列各题:

表 7.1　二叉树 bt 的一种存储结构

	1	2	3	4	5	6	7	8	9	10
lchild	0	0	2	3	7	5	8	0	10	1
data	j	h	f	d	b	a	c	e	g	i
rchild	0	0	0	9	4	0	0	0	0	0

（1）画出二叉树 bt 的树形表示。

（2）写出按先序、中序和后序遍历二叉树 bt 所得到的结点序列。

（3）画出二叉树 bt 的后序线索树（不带头结点）。

答：（1）二叉树 bt 的树形表示如图 7.3 所示。

（2）先序序列：abcedfhgij。

中序序列：ecbhfdjiga。

后序序列：echfjigdba。

（3）二叉树 bt 的后序序列为 echfjigdba，则后序线索树如图 7.4 所示。

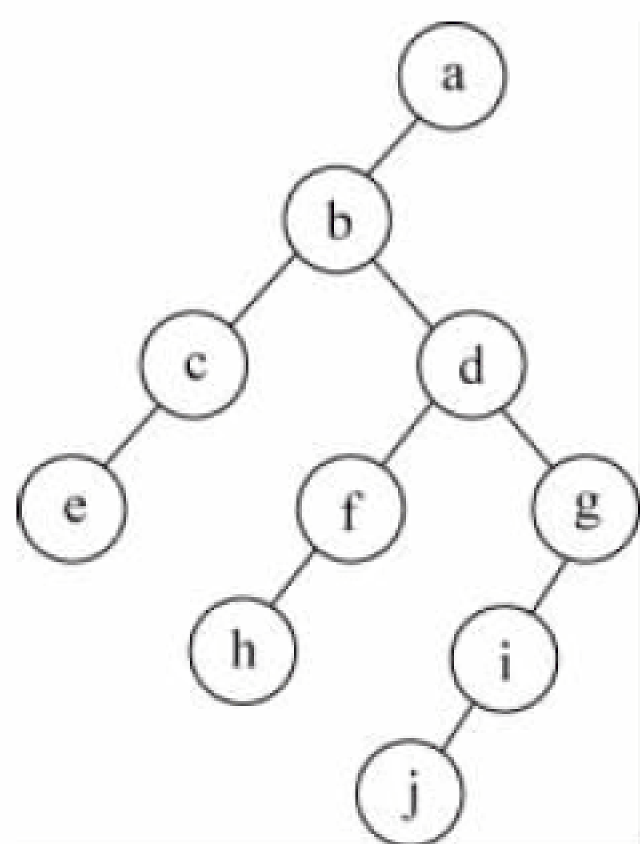


图 7.3　二叉树 bt 的逻辑结构

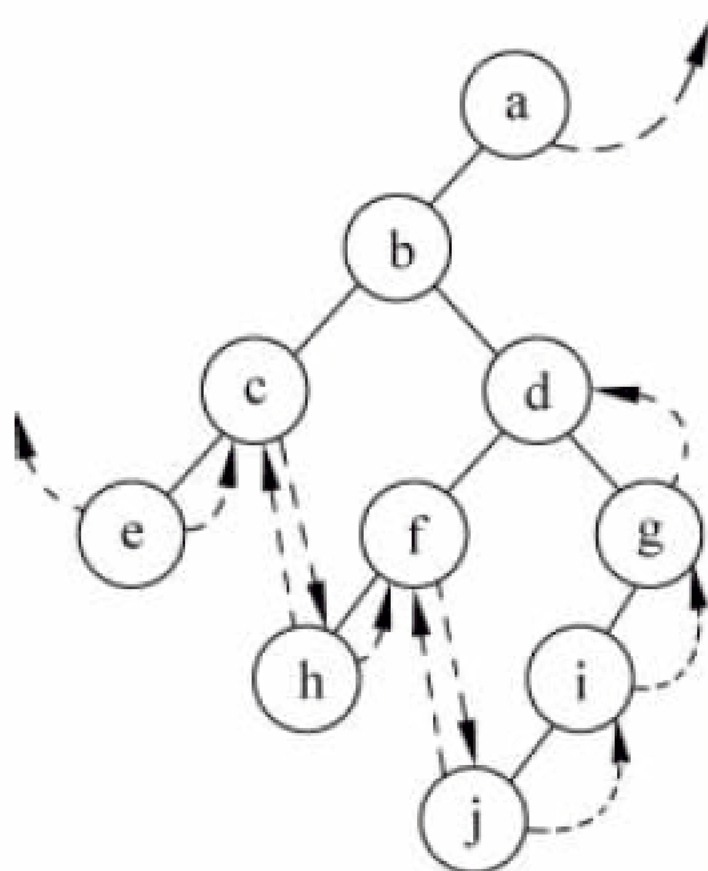


图 7.4　二叉树 bt 的后序线索化树

5. 含有 60 个叶子结点的二叉树的最小高度是多少？

答：在该二叉树中，$n_0=60$，$n_2=n_0-1=59$，$n=n_0+n_1+n_2=119+n_1$，当 $n_1=0$ 且为完全二叉树时高度最小，此时高度 $h=\lceil\log_2(n+1)\rceil=\lceil\log_2 120\rceil=7$。所以含有 60 个叶子结点的二叉树的最小高度是 7。

6. 已知一棵完全二叉树的第 6 层（设根结点为第 1 层）有 8 个叶子结点，则该完全二叉树的结点个数最多是多少？最少是多少？

答：完全二叉树的叶子结点只能在最下面两层，所以结点最多的情况是第 6 层为倒数第 2 层，即 1～6 层构成一棵满二叉树，其结点总数为 $2^6-1=63$。其中第 6 层有 $2^5=32$ 个结点，含 8 个叶子结点，则另外有 $32-8=24$ 个非叶子结点，它们中的每个结点有两个孩子结点（均为第 7 层的叶子结点），计为 48 个叶子结点。这样最多的结点个数$=63+48=111$。

结点最少的情况是第 6 层为最下层，即 1～5 层构成一棵满二叉树，其结点总数为 $2^5-1=31$，再加上第 6 层的结点，总计 $31+8=39$。这样最少的结点个数为 39。

7. 已知一棵满二叉树的结点个数为 20～40，此二叉树的叶子结点有多少个？

答：一棵高度为 h 的满二叉树的结点个数为 2^h-1，有 $20\leqslant 2^h-1\leqslant 40$。

则 $h=5$，满二叉树中的叶子结点均集中在最底层，所以叶子结点个数$=2^{5-1}=16$ 个。

8. 已知一棵二叉树的中序序列为 cbedahgijf、后序序列为 cedbhjigfa，给出该二叉树的树形表示。

答：该二叉树的构造过程和二叉树如图 7.5 所示。

9. 给定 5 个字符 a～f，它们的权值集合 $W=\{2,3,4,7,8,9\}$，试构造关于 W 的一棵哈夫曼树，求其带权路径长度 WPL 和各个字符的哈夫曼树编码。

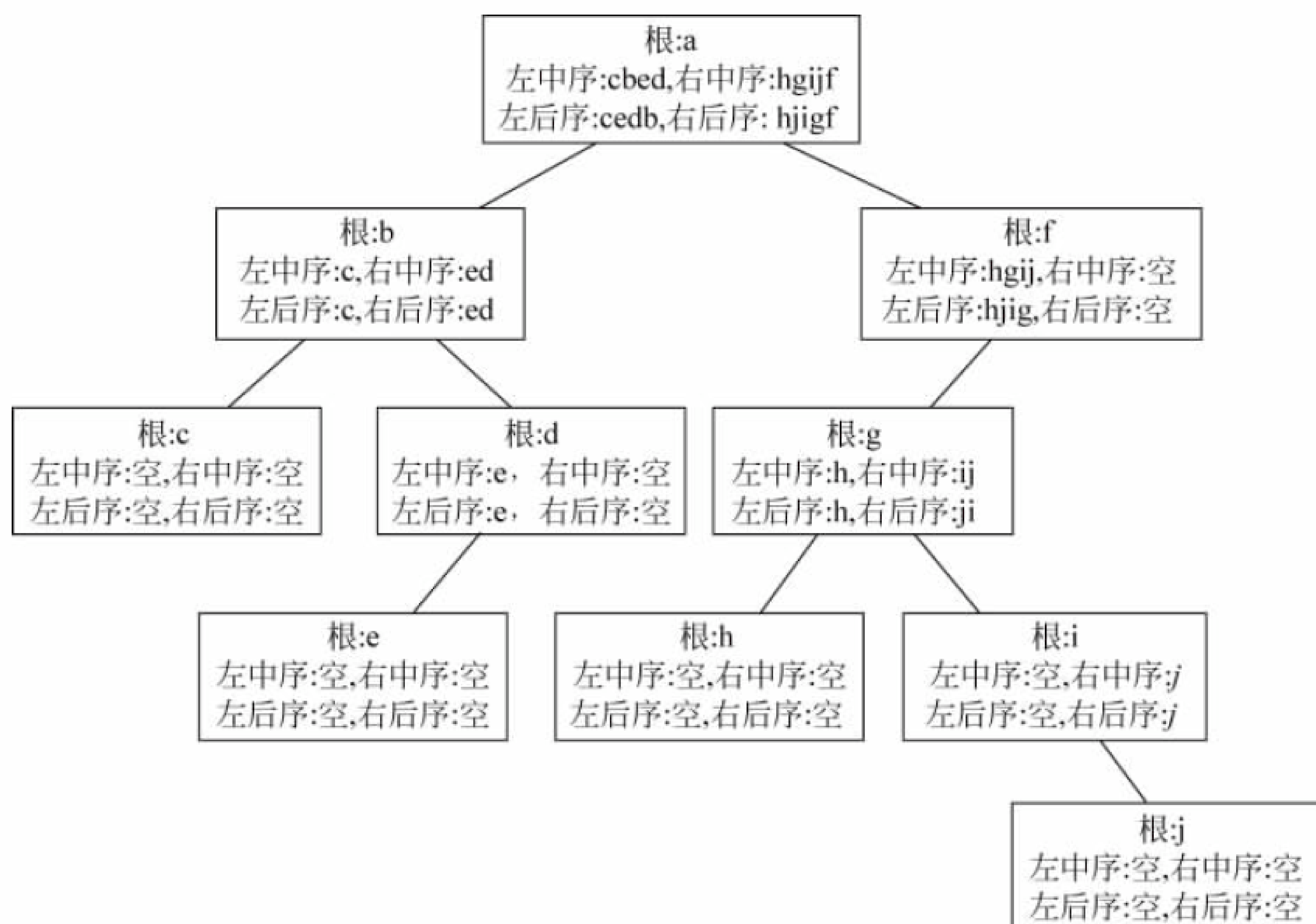


图 7.5 二叉树的构造过程

答：由权值集合 W 构建的哈夫曼树如图 7.6 所示。其带权路径长度 WPL＝(9＋7＋8)×2＋4×3＋(2＋3)×4＝80。

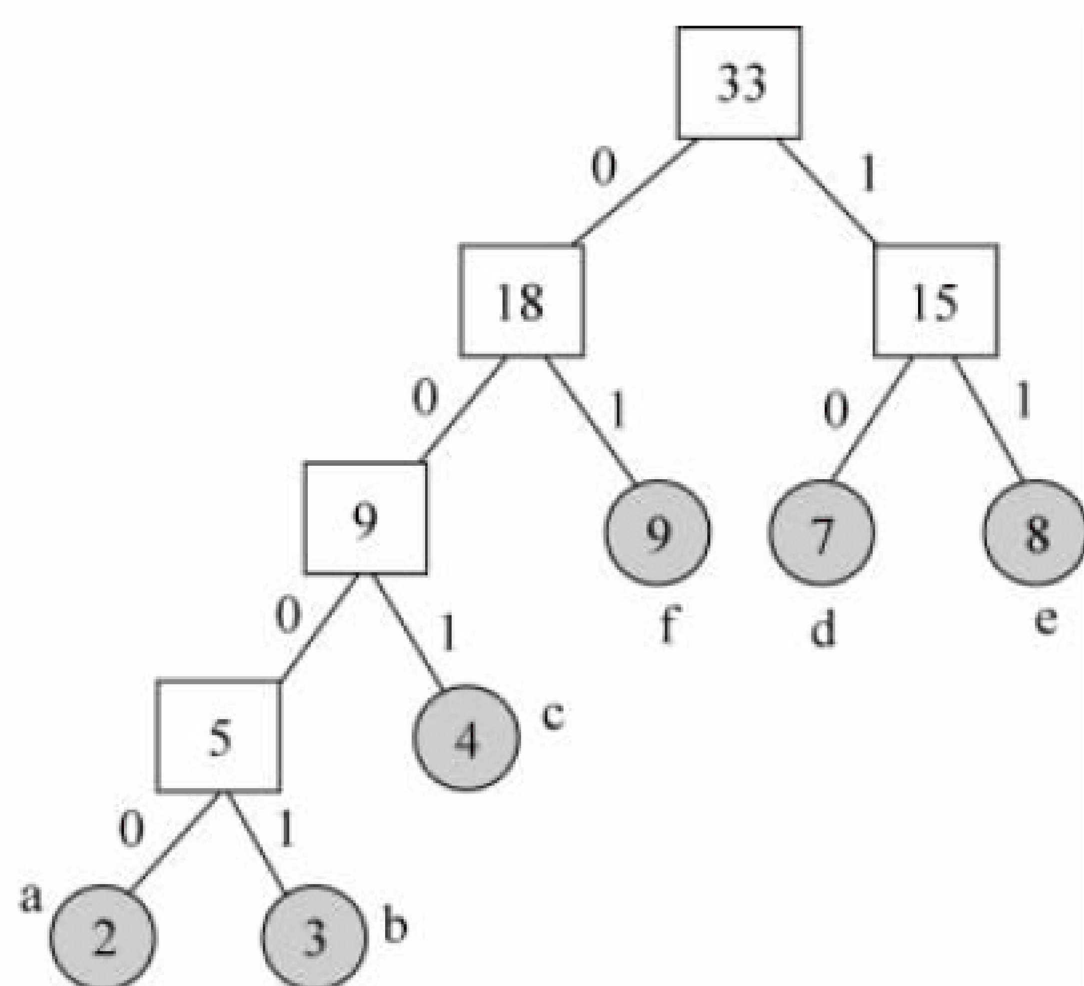


图 7.6 一棵哈夫曼树

各个字符的哈夫曼树编码：

a：0000，b：0001，c：001，d：10，e：11，f：01。

10．假设二叉树中每个结点的值为单个字符，设计一个算法，将一棵以二叉链方式存储的二叉树 b 转换成对应的顺序存储结构 a。

解：设二叉树的顺序存储结构类型为 SqBTree，先将顺序存储结构 a 中的所有元素置为'＃'(表示空结点)。将 b 转换成 a 的递归模型如下：

$$f(\mathrm{b},\mathrm{a},i)\equiv\begin{cases}\mathrm{a}[i]='\#'; & \text{当 } b=\mathrm{NULL}\\ \text{由 b 结点 data 域值建立 a}[i]\text{元素}; & \text{其他情况}\\ \quad f(\mathrm{b}\text{->}\mathrm{lchild},\mathrm{a},2*i); & \\ \quad f(\mathrm{b}\text{->}\mathrm{rchild},\mathrm{a},2*i+1) & \end{cases}$$

调用方式为 f(b,a,1)(a 的下标从 1 开始)。对应的算法如下:

```
void Ctree(BTNode *b,SqBTree a,int i)
{   if (b!= NULL)
    {   a[i] = b->data;
        Ctree(b->lchild,a,2*i);
        Ctree(b->rchild,a,2*i+1);
    }
    else a[i] = '#';
}
```

11. 假设二叉树中的每个结点值为单个字符,采用顺序存储结构存储。设计一个算法,求二叉树 t 中的叶子结点个数。

解:用 i 遍历所有的结点,当 i 大于等于 MaxSize 时,返回 0。当 $t[i]$是空结点时返回 0;当 $t[i]$是非空结点时,若它为叶子结点,num 增 1,否则递归调用 num1=LeftNode(t,2*i)求出左子树的叶子结点个数 num1,再递归调用 num2=LeftNode(t,2*i+1)求出右子树的叶子结点个数 num2,置 num+=num1+num2,最后返回 num。对应的算法如下:

```
int LeftNode(SqBTree t,int i)
{   //i 的初值为 1
    int num1,num2,num = 0;
    if (i<MaxSize)
    {   if (t[i]!= '#')
        {   if (t[2*i] == '#' && t[2*i+1] == '#')
                num++;                  //叶子结点个数增 1
            else
            {   num1 = LeftNode(t,2*i);
                num2 = LeftNode(t,2*i+1);
                num += num1 + num2;
            }
            return num;
        }
        else return 0;
    }
    else return 0;
}
```

12. 假设二叉树中的每个结点值为单个字符,采用二叉链存储结构存储。设计一个算法,计算一棵给定二叉树 b 中的所有单分支结点个数。

解:计算一棵二叉树的所有单分支结点个数的递归模型 $f(b)$如下。

$$f(b)=\begin{cases}0 & \text{若 } b=\text{NULL}\\ f(b->\text{lchild})+f(b->\text{rchild})+1 & \text{若 } b \text{ 节点为单分支}\\ f(b->\text{lchild})+f(b->\text{rchild}) & \text{其他情况}\end{cases}$$

对应的算法如下：

```
int SSonNodes(BTNode *b)
{   int num1,num2,n;
    if (b==NULL)
        return 0;
    else if ((b->lchild==NULL && b->rchild!=NULL) ||
        (b->lchild!=NULL && b->rchild==NULL))
        n=1;                        //为单分支结点
    else
        n=0;                        //其他结点
    num1 = SSonNodes(b->lchild);    //递归求左子树中的单分支结点数
    num2 = SSonNodes(b->rchild);    //递归求右子树中的单分支结点数
    return (num1 + num2 + n);
}
```

上述算法采用的是先序遍历的思路。

13. 假设二叉树中的每个结点值为单个字符，采用二叉链存储结构存储。设计一个算法，求二叉树 b 中最小值的结点值。

解：设 $f(b,\min)$是在二叉树 b 中寻找最小结点值 min，其递归模型如下：

$f(b,\min)$≡不做任何事件　　　　　　若 b=NULL

$f(b,\min)$≡当 b—> data < min 时置 $min=b$—> data；　其他情况

　　$f(b$—> lchild,min)；$f(b$—> rchild,min)；

对应的算法如下：

```
void FindMinNode(BTNode *b,char &min)
{   if (b->data<min)
        min=b->data;
    FindMinNode(b->lchild,min);        //在左子树中找最小结点值
    FindMinNode(b->rchild,min);        //在右子树中找最小结点值
}
void MinNode(BTNode *b)                //输出最小结点值
{   if (b!=NULL)
    {   char min=b->data;
        FindMinNode(b,min);
        printf("Min= %c\n",min);
    }
}
```

14. 假设二叉树中的每个结点值为单个字符，采用二叉链存储结构存储。设计一个算法，将二叉链 $b1$ 复制到二叉链 $b2$ 中。

解：当 $b1$ 为空时，置 $b2$ 为空树。当 $b1$ 不为空时，建立 $b2$ 结点（$b2$ 为根结点），置 $b2$—> data=$b1$—> data；递归调用 Copy($b1$—> lchild,$b2$—> lchild)，由 $b1$ 的左子树建立 $b2$ 的左子树；递归调用 Copy($b1$—> rchild,$b2$—> rchild)，由 $b1$ 的右子树建立 $b2$ 的右子树。对应的算法如下：

```
void Copy(BTNode *b1,BTNode *&b2)
{   if (b1 == NULL)
        b2 = NULL;
    else
    {   b2 = (BTNode *)malloc(sizeof(BTNode));
        b2->data = b1->data;
        Copy(b1->lchild,b2->lchild);
        Copy(b1->rchild,b2->rchild);
    }
}
```

15．假设二叉树中的每个结点值为单个字符，采用二叉链存储结构存储。设计一个算法，求二叉树 b 中第 k 层上的叶子结点个数。

解：采用先序遍历方法，当 b 为空时返回 0。置 num 为 0。若 b 不为空，当前结点的层次为 k，并且 b 为叶子结点，则 num 增 1，递归调用 num1＝LevelkCount(b—> lchild，k，h＋1)求出左子树中第 k 层的结点个数 num1，递归调用 num2＝LevelkCount(b—> rchild，k，h＋1)求出右子树中第 k 层的结点个数 num2，置 num＋＝num1＋num2，最后返回 num。对应的算法如下：

```
int LevelkCount(BTNode *b,int k,int h)
{   //h 的初值为 1
    int num1,num2,num = 0;
    if (b!= NULL)
    {   if (h == k && b->lchild == NULL && b->rchild == NULL)
            num++;
        num1 = LevelkCount(b->lchild,k,h+1);
        num2 = LevelkCount(b->rchild,k,h+1);
        num += num1 + num2;
        return num;
    }
    return 0;
}
int Levelkleft(BTNode *b,int k          //返回二叉树 b 中第 k 层上的叶子结点个数
{
    return LevelkCount(b,k,1);
}
```

16．假设二叉树中的每个结点值为单个字符，采用二叉链存储结构存储。设计一个算法，判断值为 x 的结点与值为 y 的结点是否互为兄弟，假设这样的结点值是唯一的。

解：采用先序遍历方法，当 b 为空时直接返回 false；否则，若当前结点 b 是双分支结点，且有两个互为兄弟的结点 x、y，则返回 true；否则递归调用 flag＝Brother(b—> lchild，x，y)，求出 x、y 在左子树中是否互为兄弟，若 flag 为 true，则返回 true；否则递归调用 Brother(b—> rchild，x，y)，求出 x、y 在右子树中是否互为兄弟，并返回其结果。对应的算法如下：

```
bool Brother(BTNode *b,char x,char y)
{   bool flag;
```

```
        if (b == NULL)
            return false;
        else
        {   if (b -> lchild!= NULL && b -> rchild!= NULL)
            {   if ((b -> lchild -> data == x && b -> rchild -> data == y) ||
                    (b -> lchild -> data == y && b -> rchild -> data == x))
                    return true;
            }
            flag = Brother(b -> lchild,x,y);
            if (flag == true)
                return true;
            else
                return Brother(b -> rchild,x,y);
        }
    }
```

17. 假设二叉树中的每个结点值为单个字符，采用二叉链存储结构存储。设计一个算法，采用先序遍历方法求二叉树 b 中值为 x 的结点的子孙结点，假设值为 x 的结点是唯一的。

解：设计 Output(p)算法输出以 p 为根结点的所有结点。首先在二叉树 b 中查找值为 x 的结点，当前 b 结点是这样的结点，调用 Output(b—> lchild)输出其左子树中的所有结点，调用 Output(b—> rchild)输出其右子树中的所有结点，并返回；否则递归调用 Child(b—> lchild,x)在左子树中查找值为 x 的结点，递归调用 Child(b—> rchild,x)在右子树中查找值为 x 的结点。对应的算法如下：

```
void Output(BTNode * p)                    //输出以 p 为根结点的子树
{   if (p!= NULL)
    {   printf(" %c ",p -> data);
        Output(p -> lchild);
        Output(p -> rchild);
    }
}
void Child(BTNode * b,char x)              //输出 x 结点的子孙结点
{   if (b!= NULL)
    {   if (b -> data == x)
        {   if (b -> lchild!= NULL)
                Output(b -> lchild);
            if (b -> rchild!= NULL)
                Output(b -> rchild);
            return;
        }
        Child(b -> lchild,x);
        Child(b -> rchild,x);
    }
}
```

18. 假设二叉树采用二叉链存储结构，设计一个算法把二叉树 b 的左、右子树进行交换，要求不破坏原二叉树，并用相关数据进行测试。

解：交换二叉树的左、右子树的递归模型如下。

$$f(b,t) \equiv \begin{cases} t=\text{NULL} & \text{若 } b=\text{NULL} \\ \text{复制根结点 } b \text{ 产生结点 } t\text{；} & \text{其他情况} \\ \quad f(b\text{->lchild},t1)\text{；}\quad f(b\text{->rchild},t2)\text{；} & \\ \quad t\text{->lchild}=t2\text{；}\quad t\text{->rchild}=t1 & \end{cases}$$

对应的算法如下(算法返回左、右子树交换后的二叉树)：

```
#include "btree.cpp"                    //二叉树基本运算算法
BTNode *Swap(BTNode *b)
{   BTNode *t, *t1, *t2;
    if (b == NULL)
        t = NULL;
    else
    {   t = (BTNode *)malloc(sizeof(BTNode));
        t->data = b->data;              //复制产生根结点 t
        t1 = Swap(b->lchild);
        t2 = Swap(b->rchild);
        t->lchild = t2;
        t->rchild = t1;
    }
    return t;
}
```

或者设计成以下算法(算法产生左、右子树交换后的二叉树 $b1$)：

```
void Swap1(BTNode *b,BTNode *&b1)
{   if (b == NULL)
        b1 = NULL;
    else
    {   b1 = (BTNode *)malloc(sizeof(BTNode));
        b1->data = b->data;          //复制产生根结点 b1
        Swap1(b->lchild,b1->rchild);
        Swap1(b->rchild,b1->lchild);
    }
}
```

设计以下主函数：

```
int main()
{   BTNode *b, *b1;
    CreateBTree(b,"A(B(D(,G)),C(E,F))");
    printf("交换前的二叉树:");DispBTree(b);printf("\n");
    b1 = Swap(b);
    printf("交换后的二叉树:");DispBTree(b1);printf("\n");
    DestroyBTree(b);
```

```
    DestroyBTree(b1);
    return 1;
}
```

程序的执行结果如下：

```
交换前的二叉树:A(B(D(,G)),C(E,F))
交换后的二叉树:A(C(F,E),B(,D(G)))
```

19. 假设二叉树采用二叉链存储结构，设计一个算法判断一棵二叉树 b 的左、右子树是否同构。

解：判断二叉树 $b1$、$b2$ 是否同构的递归模型如下。

$$f(b1,b2)=\begin{cases}\text{true} & b1=b2=\text{NULL}\\ \text{false} & \text{若 } b1\text{、}b2 \text{ 中有一个为空，另一个不为空}\\ f(b1->\text{lchild},b2->\text{lchild}) & \text{其他情况}\\ \quad \& \; f(b1->\text{rchild},b2->\text{rchild}) & \end{cases}$$

对应的算法如下：

```
bool Symm(BTNode * b1,BTNode * b2)              //判断二叉树 b1 和 b2 是否同构
{   if (b1 == NULL && b2 == NULL)
        return true;
    else if (b1 == NULL || b2 == NULL)
        return false;
    else
        return (Symm(b1->lchild,b2->lchild) & Symm(b1->rchild,b2->rchild));
}
bool Symmtree(BTNode * b)                       //判断二叉树的左、右子树是否同构
{   if (b == NULL)
        return true;
    else
        return Symm(b->lchild,b->rchild);
}
```

20. 假设二叉树以二叉链存储，设计一个算法判断一棵二叉树 b 是否为完全二叉树。

解：根据完全二叉树的定义，对完全二叉树按照从上到下、从左到右的次序遍历(层次遍历)应该满足以下条件。

(1) 某结点没有左孩子，则一定无右孩子。

(2) 若某结点缺左或右孩子(一旦出现这种情况，置 bj=false)，则其所有后继一定无孩子。

若不满足上述任何一条，均不为完全二叉树(cm=true 表示是完全二叉树，cm=false 表示不是完全二叉树)。对应的算法如下：

```
bool CompBTree(BTNode * b)
{   BTNode * Qu[MaxSize], * p;               //定义一个队列,用于层次遍历
```

```
    int front = 0,rear = 0;                //环形队列的队头、队尾指针
    bool cm = true;                        //cm 为真表示二叉树为完全二叉树
    bool bj = true;                        //bj 为真表示到目前为止所有结点均有左、右孩子
    if (b == NULL) return true;            //空树当成特殊的完全二叉树
    rear++;
    Qu[rear] = b;                          //根结点进队
    while (front!= rear)                   //队列不空
    {   front = (front + 1) % MaxSize;
        p = Qu[front];                     //出队结点 p
        if (p -> lchild == NULL)           //p 结点没有左孩子
        {   bj = false;                    //出现结点 p 缺左孩子的情况
            if (p -> rchild!= NULL)        //没有左孩子但有右孩子,违反(1)
                cm = false;
        }
        else                               //p 结点有左孩子
        {   if (!bj) cm = false;           //bj 为假而结点 p 还有左孩子,违反(2)
            rear = (rear + 1) % MaxSize;
            Qu[rear] = p -> lchild;        //左孩子进队
            if (p -> rchild == NULL)
                bj = false;                //出现结点 p 缺右孩子的情况
            else                           //p 有左、右孩子,则继续判断
            {   rear = (rear + 1) % MaxSize;
                Qu[rear] = p -> rchild;    //将 p 结点的右孩子进队
            }
        }
    }
    return cm;
}
```

7.3 补充练习题及参考答案



7.3.1 单项选择题

1. 对于一棵具有 n 个结点、度为 4 的树来说，________。

   A. 树的高度最多是 $n-3$　　B. 树的高度最多是 $n-4$

   C. 第 i 层上最多有 $4(i-1)$ 个结点　　D. 至少在某一层上正好有 4 个结点

答：这样的树中至少有一个结点的度为 4，也就是说，至少有一层中有 4 个或以上的结点，因此树的高度最多是 $n-3$。本题的答案为 A。

2. 度为 4、高度为 h 的树________。

   A. 至少有 $h+3$ 个结点　　B. 最多有 4^h-1 个结点

   C. 最多有 $4h$ 个结点　　D. 至少有 $h+4$ 个结点

答：与上小题分析相同，本题的答案为 A。

3. 对于一棵具有 n 个结点、度为 4 的树来说，树的高度至少是________。

A. $\lceil \log_4(2n) \rceil$　　B. $\lceil \log_4(3n-1) \rceil$

C. $\lceil \log_4(3n+1) \rceil$　　D. $\lceil \log_4(2n+1) \rceil$

答：由树的性质 4 可知，具有 n 个结点的 m 次树的最小高度为 $\lceil \log_m(n(m-1)+1) \rceil$。这里 $m=4$，因此最小高度为 $\lceil \log_4(3n+1) \rceil$。本题的答案为 C。

4. 在一棵 3 次树中度为 3 的结点数为两个，度为 2 的结点数为一个，度为 1 的结点数为两个，则度为 0 的结点数为________个。

A. 4　　B. 5　　C. 6　　D. 7

答：$n_3=2$，$n_2=1$，$n_1=2$，$n=n_3+n_2+n_1+n_0=5+n_0$，$n=$度之和$+1=3n_3+2n_2+n_1+1=11$，所以 $n_0=11-5=6$。本题的答案为 C。

5. 若一棵有 n 个结点的树，其中所有分支结点的度均为 k，该树中的叶子结点个数是________。

A. $n(k-1)/k$　　B. $n-k$　　C. $(n+1)/k$　　D. $(nk-n+1)/k$

答：$m=k$，有 $n=n_0+n_k$，度之和$=n-1=kn_k$，$n_k=(n-1)/k$，所以 $n_0=n-n_k=n-(n-1)/k=(nk-n+1)/k$。本题的答案为 D。

6. 若 3 次树中有 a 个度为 1 的结点、b 个度为 2 的结点、c 个度为 3 的结点，则该树有________个叶子结点。

A. $1+2b+3c$　　B. $1+2b+3c$　　C. $2b+3c$　　D. $1+b+2c$

答：$n=n_0+n_1+n_2+n_3=n_0+a+b+c$，$n=$度之和$+1=n_1+2n_2+3n_3+1=a+2b+3c+1$，所以，$n_0=b+2c+1$，总结点数 $n=a+2b+3c+1$。本题的答案为 D。

7. 假设每个结点值为单个字符，而一棵树的层次遍历序列为 ABCDEFGHIJ，则其根结点的值是________。

A. A　　B. B　　C. J　　D. 以上都不对

答：树的层次遍历过程中访问的第一个结点是根结点，本题的答案为 A。

8. 用双亲存储结构表示树，其优点之一是比较方便________。

A. 找指定结点的双亲结点　　B. 找指定结点的孩子结点

C. 找指定结点的兄弟结点　　D. 判断某结点是不是叶子结点

答：A。

9. 用孩子链存储结构表示树，其优点之一是________比较方便。

A. 判断两个指定结点是不是兄弟　　B. 找指定结点的双亲

C. 判断指定结点在第几层　　D. 计算指定结点的度数

答：在树的孩子链存储结构中，每个结点有指向所有孩子结点的指针，所以很容易计算其孩子结点个数(度数)。本题的答案为 D。

10. 一棵度为 10、结点个数为 $n(n>100)$的树采用孩子链存储结构时，其中非空指针域数占总指针域数的比例约为________。

A. 5%　　B. 10%　　C. 20%　　D. 50%

答：在度为 10 树的孩子链存储结构中，每个结点的指针域个数为 10，共有 $10n$ 个指针域，其中非空的指针域个数等于分支数，即 $n-1$，其余为空指针域，所以非空指针域数占总指针域数的比例$=(n-1)/(10n)\approx 10\%$。本题的答案为 B。

11. 如果在树的孩子兄弟链存储结构中有6个空的左指针域，7个空的右指针域，5个结点左、右指针域都为空，则该树中叶子结点的个数________。

A. 有7个　　B. 有6个　　C. 有5个　　D. 不能确定

答：在树的孩子兄弟链存储结构中，左指针域指向第一个孩子结点，右指针域指向右兄弟结点。该树有6个空的左指针域，说明有6个结点没有任何孩子，则为叶子结点。本题的答案为B。

12. 有一棵3次树，其中$n_3=2$，$n_2=2$，$n_1=1$，当该树采用孩子兄弟链存储结构时，其中非空指针域数占总指针域数的比例约为________。

A. 10%　　B. 45%　　C. 70%　　D. 90%

答：$m=3$，$n=n_0+n_1+n_2+n_3=n_0+5$，而$n-1=n_1+2n_2+3n_3=11$，所以$n=12$，$n_0=7$。当采用孩子兄弟链存储结构时，每个结点有两个指针域，一个指向孩子，一个指向兄弟，总指针域个数为24。指向孩子或者兄弟的非空指针域个数$=n-1=11$，所以非空指针域数占总指针域数的比例$=11/24\approx45\%$。本题的答案为B。

13. 设森林F中有3棵树，第一、第二和第三棵树的结点个数分别为m_1、m_2和m_3。与森林F对应的二叉树根结点的右子树上的结点个数是________。

A. m_1　　B. m_1+m_2　　C. m_3　　D. m_2+m_3

答：对应的二叉树根结点的右子树上的结点均由第二和第三棵树上的结点转换得到。本题的答案为D。

14. 设F是一个森林，B是由F变换的二叉树。若F中有m个分支结点，则B中右指针域为空的结点有________个。

A. $m-1$　　B. m　　C. $m+1$　　D. $m+2$

答：F中的每个分支结点都有一个最右孩子结点，这个最右孩子结点在B中右指针域为空，同时根结点的右指针域也为空，所以B中共有$m+1$个右指针域为空的结点。本题的答案为C。

15. 设森林F对应的二叉树为B，它有m个结点，B的根为p，p的右子树结点个数为n，森林F中第一棵树的结点个数是________。

A. $m-n$　　B. $m-n-1$

C. $n+1$　　D. 条件不足，无法确定

答：森林F中的第一棵树转换成二叉树p及p的左子树。本题的答案为A。

16. 如果将一棵有序树T转换为二叉树B，那么T中结点的先根遍历序列就是B中结点的________序列。

A. 先序　　B. 中序　　C. 后序　　D. 层次序

答：若树T的根为N，它的子树为$T_1,T_2,\cdots,T_m$，其先根序列为$NT_1T_2\cdots T_m$。树T转换成二叉树B的过程如图7.7所示（T_i转换为B_i），B_1为N的左子树中结点，$B_2,\cdots,B_m$的所有结点都在B_1根结点的右子树中，T的$NT_1T_2\cdots T_m$序列在B中遍历过程是先根结点、再左子树，对应B的先序序列。本题的答案为A。

17. 如果将一棵有序树T转换为二叉树B，那么T中结点的后根遍历序列就是B中结点的________序列。

A. 先序　　B. 中序　　C. 后序　　D. 层次序

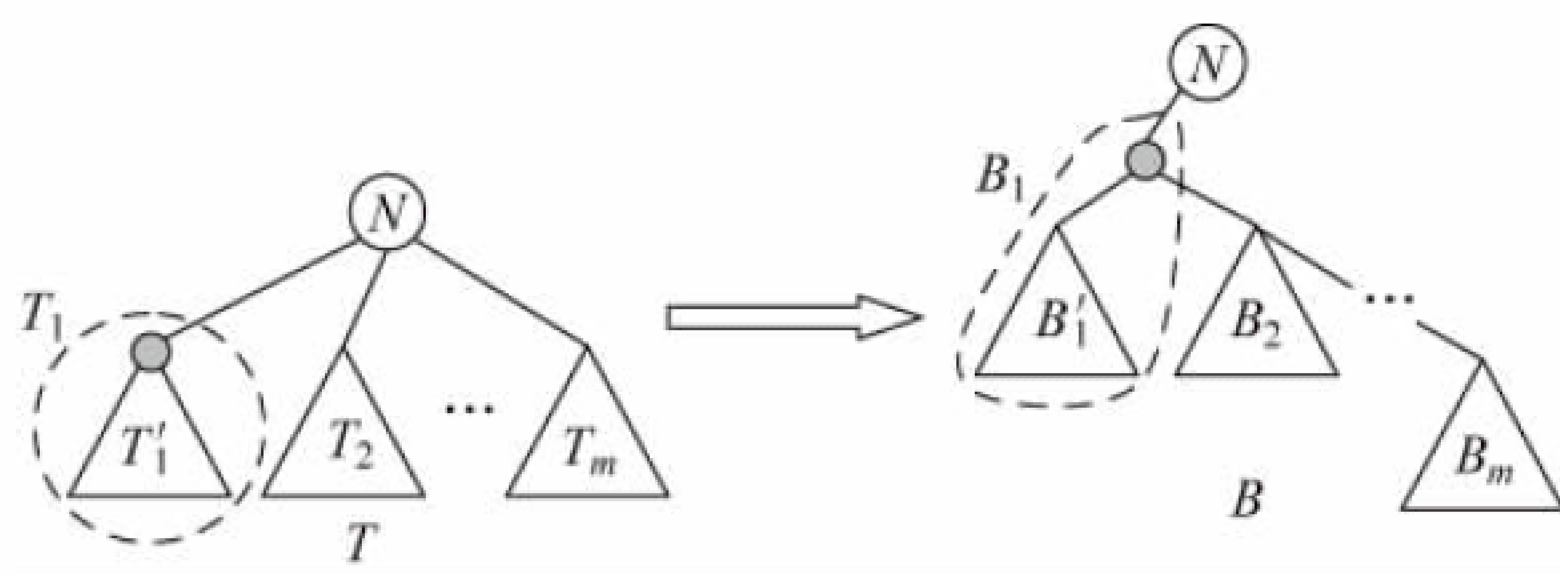


图 7.7 T 转换为 B

答：若树 T 的根为 N，它的子树为 T_1、T_2、…、T_m，其后根序列为 $T_1T_2\cdots T_mN$。树 T 转换成二叉树 B 的过程如图 7.7 所示（T_i 转换为 B_i），B_1 为 N 的左子树中结点，B_2、…、B_m 的所有结点都在 B_1 根结点的右子树中，T 的 $T_1T_2\cdots T_mN$ 序列在 B 中遍历过程是先左子树，再根结点。又显然不会是 B 的后序序列，因为在 B 的后序遍历中，B_2 中结点会在 B_1 根结点之前访问，所以只能是 B 的中序序列。本题的答案为 B。

18. 如果将一棵有序树 T 转换为二叉树 B，那么 T 中结点的层次序列对应 B 的________序列。

A. 先序遍历　　B. 中序遍历　　C. 层次遍历　　D. 以上都不对

答：由于 T 中的兄弟变为 B 中的右孩子，改变为父子关系，所以 T 中结点的层次序列与 B 的先序、中序、后序和层次序列都没有对应关系。本题的答案为 D。

19. 二叉树若用顺序方法存储，则下列 4 种运算中________最容易实现。

A. 先序遍历二叉树

B. 判断两个结点值分别为 x、y 的结点是不是在同一层上

C. 层次遍历二叉树

D. 求结点值为 x 的结点的所有孩子

答：直接顺序扫描存储二叉树的数组即得到层次遍历二叉树序列。本题的答案为 C。

20. 二叉树和度为 2 的树的相同之处包括________。

A. 每个结点都有一个或两个孩子结点　　B. 至少有一个根结点

C. 至少有一个度为 2 的结点　　D. 每个结点最多只有一个双亲结点

答：D。二叉树和度为 2 的树都属于树形结构，其中每个结点最多只有一个双亲结点。

21. 一棵完全二叉树上有 1001 个结点，其中叶子结点的个数是________。

A. 250　　B. 501　　C. 254　　D. 505

答：由二叉树的性质知 $n_0=n_2+1$，且完全二叉树的 $n_1=0$ 或 1；已知二叉树的总结点数 $n=n_0+n_1+n_2$，即有 $n=2n_0+n_1-1$；将总结点数 $n=1001$ 代入得 $1001=2n_0+n_1-1$，因 1001 为奇数，故 $n_1=0$，得到 $n_0=501$。本题的答案为 B。

22. 一棵有 124 个叶子结点的完全二叉树最多有________个结点。

A. 247　　B. 248　　C. 249　　D. 250

答：由 $n_0=n_2+1$ 可知 $n_2=123$；$n=n_2+n_1+n_0=247+n_1$，在完全二叉树中，$n_1=0$ 或 1，所以 n 的最大值为 $247+1=248$，故选 B。

23. 在一棵具有 n 个结点的完全二叉树中，分支结点的最大编号为________。

A. $\lfloor (n+1)/2 \rfloor$　　B. $\lfloor (n-1)/2 \rfloor$　　C. $\lceil n/2 \rceil$　　D. $\lfloor n/2 \rfloor$

答：D。

24. 在高度为 h 的完全二叉树中，________。

A. 度为 0 的结点都在第 h 层上

B. 第 $i(1\leqslant i\leqslant h)$层上结点都是度为 2 的结点

C. 第 $i(1\leqslant i\leqslant h-1)$层上有 2^{i-1}个结点

D. 不存在度为 1 的结点

答：在高度为 h 的完全二叉树中，第 1 层～第 $h-1$ 层构成一个满二叉树，在满二叉树中第 i 层上有 2^{i-1}个结点。本题的答案为 C。

25. 每个结点的度或者为 0 或者为 2 的二叉树称为正则二叉树，对于 n 个结点的正则二叉树来说，它的最大高度是________。

A. $\lceil \log_2 n \rceil$　　B. $(n-1)/2$　　C. $\lceil \log_2(n+1) \rceil$　　D. $(n+1)/2$

答：最大高度的正则二叉树是这样的二叉树，第 1 层有一个结点，第 2 层～第 h 层均有两个结点，因此 $2(h-1)+1=n$，即 $h=(n+1)/2$。本题的答案为 D。

26. 若一棵二叉树具有 10 个度为 2 的结点、5 个度为 1 的结点，则度为 0 的结点个数是________。

A. 9　　B. 11　　C. 15　　D. 不确定

答：$n_2=10$，$n_0=n_2+1=11$。本题的答案为 B。

27. 若二叉树的中序序列是 abcdef，且 c 为根结点，则________。

A. 结点 c 有两个孩子　　B. 二叉树有两个度为 0 的结点

C. 二叉树的高度为 5　　D. 以上都不对

答：中序序列是 abcdef，则 ab 为结点 c 的左子树的中序序列，def 为结点 c 的右子树的中序序列，说明结点 c 既有左子树又有右子树。本题的答案为 A。

28. 在任何一棵二叉树中，如果结点 a 有左孩子 b、右孩子 c，则在结点的先序序列、中序序列、后序序列中，________。

A. 结点 b 一定在结点 a 的前面　　B. 结点 a 一定在结点 c 的前面

C. 结点 b 一定在结点 c 的前面　　D. 结点 a 一定在结点 b 的前面

答：在先序遍历、中序遍历和后序遍历中都是先遍历左子树，再遍历右子树，所以结点 b 一定在结点 c 的前面访问。本题的答案为 C。

29. 如果一棵二叉树的先序序列是…a…b…，中序序列是…b…a…，则________。

A. 结点 a 和结点 b 分别在某结点的左子树和右子树中

B. 结点 b 在结点 a 的右子树中

C. 结点 b 在结点 a 的左子树中

D. 结点 a 和结点 b 分别在某结点的两棵非空子树中

答：先序序列是…a…b…，则 b 在 a 的左子树中或者 b 在 a 的某个祖先 x 的右子树中，中序序列是…b…a…，则 b 不可在 a 的某个祖先 x 的右子树中，即 b 只能在 a 的左子树中。本题的答案为 C。

30. 设 a、b 为一棵二叉树上的两个结点，在中序序列时，a 在 b 之前的条件是________。

A. a 在 b 的右方　　B. a 是 b 的祖先

C. a 在 b 的左方　　D. a 是 b 的子孙

答：中序遍历时，先遍历左子树，再访问根结点，最后遍历右子树。a 在 b 前，则 a 在 b 的左子树中或 b 在 a 的右子树中，或者 a 在某棵子树的左子树中而 b 在其右子树中，这都表示 a 在 b 的左方。本题的答案为 C。

31. 如果在一棵二叉树的先序序列、中序序列和后序序列中，结点 a、b 的位置都是 a 在前、b 在后(即形如…a…b…)，则________。

A. a、b 可能是兄弟　　B. a 可能是 b 的双亲

C. a 可能是 b 的孩子　　D. 不存在这样的二叉树

答：图 7.8 所示的二叉树中 a 和 b 结点就满足本题的条件。本题的答案为 A。

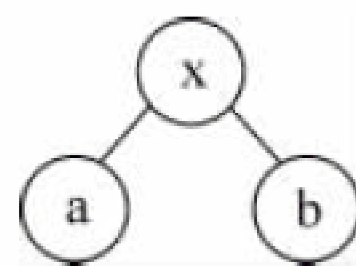


图 7.8　一棵二叉树

32. 若二叉树采用二叉链存储结构，如果要交换其所有分支结点的左、右子树位置，利用________遍历方法最合适。

A. 先序　　B. 中序

C. 后序　　D. 按层次

答：先对根结点的左、右子树进行交换，再交换根结点的左、右指针值，这是后序遍历的思路。本题的答案为 C。

33. 关于非空二叉树的后序序列以下说法正确的是________。

A. 后序序列的最后一个结点是根结点

B. 后序序列的最后一个结点一定是叶子结点

C. 后序序列的第一个结点一定是叶子结点

D. 以上都不对

答：A。

34. 某二叉树的先序序列和后序序列正好相反，则该二叉树一定是________。

A. 空或只有一个结点　　B. 完全二叉树

C. 二叉排序树　　D. 高度等于其结点数

答：二叉树的先序序列是 NLR(N 为根结点，L 为左子树，R 为右子树)，后序序列 LRN，要使 NLR＝NRL，则 L 为空或 R 为空，这样的二叉树每层只有一个结点，即高度等于其结点数。本题的答案为 D。

35. 若一棵二叉树的先序序列和后序遍历分别是 1、2、3、4 和 4、3、2、1，则该二叉树的中序序列不会是________。

A. 1、2、3、4　　B. 2、3、4、1　　C. 3、2、4、1　　D. 4、3、2、1

答：将先序序列 1、2、3、4 与某个中序序列构造出一棵二叉树，再看其后序序列是否为 4、3、2、1。当选项为 C 时，构造出的二叉树的后序序列为 3、4、2、1。本题的答案为 C。

36. 某二叉树是由一个森林转换而来，其层次序列为 ABCDEFGHI，中序序列为 DGIBAEHCF，将其还原为森林，该森林是由________棵树构成的。

A. 1　　B. 2　　C. 3　　D. 无法确定

答：由二叉树的层次序列和中序序列构造出唯一的二叉树，其中根结点有两个右下孩子，所以还原为森林后对应 3 棵树。本题的答案为 C。

37. 一棵二叉树的先序序列为 ABCDEFG，它的中序序列可能是________。

A. CABDEFG　　B. ABCDEFG　　C. DACEFBG　　D. ADCFEG

答：先序序列和中序序列可以确定一棵二叉树，这里由选项 A、C 和 D 的中序序列无法确定一棵二叉树。本题的答案为 B。

38. 在中序线索二叉树(带头结点)中，p 结点的左子树为空的充要条件是________。

A. p->lchild==NULL　　B. p->ltag==1

C. p->ltag==1 且 p->lchild==NULL　　D. 以上都不对

答：B。在带头结点的中序线索二叉树中所有指针域非空，p->ltag==1 表示左指针为线索，即原来右子树为空。

39. 在 n 个结点的线索二叉树中(不计头结点)，线索的数目为________。

A. $n-1$　　B. n　　C. $n+1$　　D. $2n$

答：n 个结点的二叉树中有 $n+1$ 个空指针，它们都转化为存放线索，所以线索的数目为 $n+1$。本题的答案为 C。

40. 若度为 m 的哈夫曼树(其中只有度为 m 的结点和叶子结点)中，其叶子结点个数为 n，则非叶子结点的个数为________。

A. $n-1$　　B. $\lfloor n/m \rfloor-1$

C. $\lceil (n-1)/(m-1) \rceil$　　D. $\lceil n/(m-1) \rceil-1$

答：在度为 m 的哈夫曼树中，设度为 m 的结点个数为 n_m，结点总数 $=n+n_m$，所有结点度之和 = 结点总数 $-1=n+n_m-1$(这样的哈夫曼树也是一棵树，满足"所有结点度之和 = 分支数 = 结点总数 -1"的特性)，而所有结点度之和 $=m\times n_m$，即 $n+n_m-1=m\times n_m$，求出 $n_m=(n-1)/(m-1)$。本题答案为 C。

41. 设有 13 个值，用它们组成一棵哈夫曼树，则该哈夫曼树共有________个结点。

A. 13　　B. 12　　C. 26　　D. 25

答：具有 n 个叶子结点的哈夫曼树共有 $2n-1$ 个结点。本题的答案为 D。

42. 根据使用频率为 5 个字符设计的哈夫曼编码不可能是________。

A. 111,110,10,01,00　　B. 000,001,010,011,1

C. 100,11,10,1,0　　D. 001,000,01,11,10

答：在 C 中，100 和 10 冲突，即一个结点既是叶子结点又是内部结点，哈夫曼树中不可能出现这种情况。本题的答案为 C。

43. 若并查集用树表示，其中有 n 个结点，查找一个元素所属集合的算法的时间复杂度为________。

A. $O(\log_2 n)$　　B. $O(n)$　　C. $O(n^2)$　　D. $O(n\log_2 n)$

答：A。

7.3.2 填空题

1. 在树形结构的二元组表示中，如果__①__，则称结点 a 和 b 是兄弟；如果__②__，则称 a 是 b 的双亲，__③__的孩子。

答：① a 和 b 有相同的前驱结点　② a 是 b 的前驱　③ b 是 a。

2. 一棵度为 2 的树中，其结点个数最少为________。

答：3。一个度为 2 的结点有 2 个孩子，加上该结点本身，所以总结点个数最少为 3。

3. 设某棵树中结点值为单个字符，其后根遍历序列为 ABCDEFG，则根结点值

为________。

答：G。

4. 对一棵具有 n 个结点的非空树，其中所有度之和等于________。

答：$n-1$。

5. 高度为 h、度为 $m(m\geqslant 2)$ 的树中最少有 ① 个结点，最多有 ② 个结点。

答：① $h+m-1$ ② $\dfrac{m^h-1}{m-1}$。

6. 一棵含有 n 个结点的 $k(k\geqslant 2)$ 次树，可能达到的最大高度为 ① 、最小高度为 ② 。

答：① n ② $\lceil \log_k(n(k-1)+1)\rceil$。最大高度为这样的 k 次树：只有一层含有 k 个结点，其余各层均只有一个结点，此时高度为 $n-k+1$。最小高度为完全 k 次树，此时高度为 $\lceil \log_k(n(k-1)+1)\rceil$。

7. 若用孩子兄弟链存储结构来存储具有 m 个叶子结点、n 个分支结点的树，则该存储结构中有 ① 个左指针域为空的结点，有 ② 个右指针域为空的结点。

答：① m ② $n+1$。在孩子兄弟链存储结构中，只有叶子结点没有孩子，其左指针域为空。每个分支结点一定有一个最右孩子，它是没有兄弟的，除此之外，根结点也是没有兄弟的，即有 $n+1$ 个结点没有兄弟，它们的右指针域为空。

8. 有 3 个结点的不同形态二叉树有________棵。

答：5。含有 n 个结点的不同形态二叉树有 $\dfrac{1}{n+1}C_{2n}^{n}$。

9. 在高度为 $h(h\geqslant 0)$ 的二叉树中最多有 ① 个结点，最少有 ② 个结点。

答：① 2^h-1(为满二叉树时结点最多) ② h(每层只有一个结点)。

10. n 个结点的二叉树的最大高度是 ① 、最小高度是 ② 。

答：① n(每层只有一个结点) ② $\lceil \log_2(n+1)\rceil$(为完全二叉树时高度最小)。

11. n 个结点的二叉树中如果有 m 个叶子结点，则一定有 ① 个度为 1 的结点， ② 个度为 2 的结点。

答：① $n-2m+1$ ② $m-1$。由二叉树的性质 1 可知，$n_0=n_2+1$，即 $n_2=n_0-1=m-1$，$n=n_0+n_1+n_2$，则 $n_1=n-n_0-n_2=n-m-(m-1)=n-2m+1$。

12. 已知二叉树有 50 个叶子结点，则该二叉树的总结点数最少是________。

答：99。结点个数最少的情况是第一层有一个结点，2～50 层有两个结点，这样共有 50 个叶子结点，49 个非叶子结点，总有 99 个结点。也可以这样推导：$n_0=50$，$n_2=50-1=49$，$n=n_0+n_1+n_2=99+n_1$，当 $n_1=0$ 时结点个数最少，此时为 99。

13. 一共 8 层的完全二叉树至少有 ① 个结点，具有 100 个结点的完全二叉树中结点的最大层数为 ② 。

答：① 128 ② 7。8 层完全二叉树具有最少结点的情况是前 7 层为满二叉树而第 8 层仅有一个结点，即为 $2^7-1+1=128$。具有 100 个结点的完全二叉树的高度是固定的，$h=\lceil \log_2(n+1)\rceil=\lceil \log_2 101\rceil=7$，而结点的最大层数恰好等于高度。

14. 完全二叉树中结点个数为 $n(n>2)$，按层序编号(根结点编号为 1)，则编号最大的

分支结点的编号为___①___，编号最小的叶子结点编号为___②___。

答：① $\lfloor n/2 \rfloor$ ② $\lfloor n/2 \rfloor+1$。

15. 一棵含有50个结点的完全二叉树中，第6层有______个结点。

答：16。该完全二叉树的高度 $h=\lceil \log_2(n+1) \rceil=\lceil \log_2 51 \rceil=6$，第1～5层是满的，共有 $2^5-1=31$ 个结点，所以第6层有 $50-31=19$ 个结点。

16. 一棵含有 n 个结点的满二叉树有___①___个度为1的结点，___②___个分支结点和___③___个叶子结点，该满二叉树的高度为___④___。

答：① 0 ② $\lfloor n/2 \rfloor$ ③ $\lfloor n/2 \rfloor+1$ ④ $\log_2(n+1)$。

17. 在二叉树的顺序存储结构中，编号分别为 i 和 j 的两个结点处在同一层的条件是______。

答：$\lceil \log_2(i+1) \rceil=\lceil \log_2(j+1) \rceil$。编号为 i 的结点所在的层号为 $\lceil \log_2(i+1) \rceil$。

18. 设 F 是由 T_1、T_2、T_3 三棵树组成的森林，与 F 对应的二叉树为 B。已知 T_1、T_2、T_3 的结点数分别为 n_1、n_2 和 n_3，则二叉树 B 的左子树中有___①___个结点，二叉树 B 的右子树中有___②___个结点。

答：① n_1-1 ② n_2+n_3。根据森林转化为二叉树的方法可知，根结点和左子树来源于森林的第一棵树，而其余的树都在根结点的右子树上。

19. 对于高度为3的满二叉树 B，将其还原为森林 T，其中包含根结点的那棵树中有______个结点。

答：4。

20. 一棵树中结点a的第2个孩子为结点b，转换成二叉树后，a、b两结点的层次相差为______。

答：2。

21. 一棵二叉树的根结点为a，其中序序列的第一个结点是___①___，其中序序列的最后一个结点是___②___。

答：① a结点的最左下结点 ②a结点的最右下结点。

22. 若一个二叉树的叶子结点是其中序序列中的最后一个结点，则它必是该二叉树的______序列中的最后一个结点。

答：先序遍历。设结点b是中序序列中的最后一个结点，根结点是a，则b一定是a的最右下结点，在先序遍历中，以b为根结点的子树一定是最后遍历的，而b又是叶子结点，所以b必是该二叉树的先序序列中的最后一个结点。

23. 在二叉树中结点a的右孩子为结点b，那么在后序序列中必有______形式。

答：…b a…。在后序遍历中，子树的根结点最后访问，当a结点的右子树遍历完后立即访问a。

24. 二叉树中一个叶子结点a是其中序序列的第一个结点，则a结点一定是该二叉树的______序列中的第一个结点。

答：后序。a结点是中序序列的第一个结点，说明它是根结点的最左下结点。在后序遍历中，以a为根结点的子树一定是最先遍历的，而它是叶子结点，没有右孩子，所以也一定是后序序列的第一个结点。

25. 设一棵完全二叉树（每个结点值为单个字符）的顺序存储结构中存储数据元素为

abcdef,则该二叉树的先序序列为___①___、中序序列为___②___、后序序列为___③___。

答:① abdecf ② dbeafc ③ debfca。

26. 设一棵完全二叉树(每个结点值为单个字符)的先序序列为 abdecf,则该二叉树的中序序列为___①___、层次序列为___②___。

答:① dbeafc ② abcdef。

27. 二叉树的先序序列和中序序列相同的条件是________。

答:该二叉树中每个结点最多只有一个右孩子。先序序列为 NLR,中序序列为 LNR,要使 NLR=LNR,只有 L 为空或 L、R 均为空时,因此这样的二叉树每层只有一个结点,非叶子结点只有右孩子。

28. 在二叉树的非递归中序和后序遍历算法中需要用________来暂存遍历的结点。

答:栈。

29. 线索二叉树的左线索指向其___①___结点,右线索指向其___②___结点。

答:① 前驱 ② 后继。

30. 若以{4,5,6,7,8}作为叶子结点的权值构造哈夫曼树,则其带权路径长度是___①___,各结点对应的哈夫曼编码为___②___。

答:① 69 ② 010、011、10、11、00。构造的哈夫曼树如图 7.9 所示,WPL=(4+5)×3+(6+7+8)×2=69。

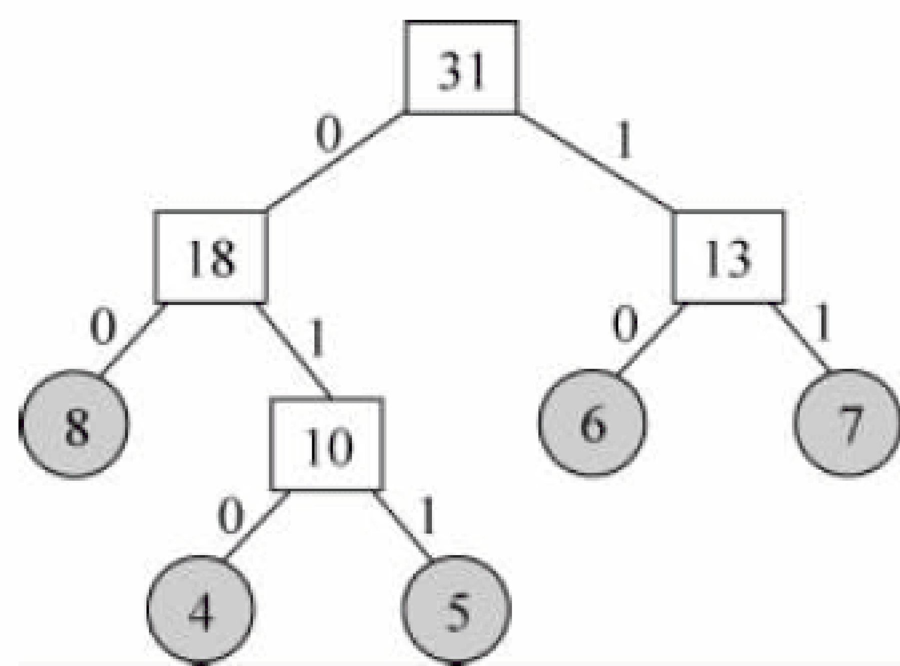


图 7.9 一棵哈夫曼树

7.3.3 判断题

1. 判断以下叙述的正确性。

(1) 树形结构中的每个结点都有一个前驱结点。

(2) 度为 m 的树中至少有一个度为 m 的结点,不存在度大于 m 的结点。

(3) 在一棵树中,处于同一层上的各结点之间都存在兄弟关系。

(4) $n(n>2)$个结点的二叉树中至少有一个度为 2 的结点。

(5) 不存在这样的二叉树:它有 n 个度为 0 的结点,$n-1$ 个度为 1 的结点,$n-2$ 个度为 2 的结点。

(6) 在任何一棵完全二叉树中,叶子结点或者和分支结点一样多,或者只比分支结点多一个。

(7) 完全二叉树中的每个结点或者没有孩子或者有两个孩子。

(8) 当二叉树中的结点数多于 1 个时,不可能根据结点的先序序列和后序序列唯一地确定该二叉树的逻辑结构。

(9) 只要知道完全二叉树中结点的先序序列就可以唯一地确定它的逻辑结构。

(10) 哈夫曼树中不存在度为 1 的结点。

(11) 在哈夫曼树中,权值相同的叶子结点都在同一层上。

(12) 在哈夫曼树中,权值较大的叶子结点一般离根结点较远。

答:(1) 错误。根结点没有前驱结点。

(2) 正确。

(3) 错误。处于同一层并且有相同双亲的各结点之间是兄弟关系。

(4) 错误。

(5) 正确。不满足 $n_0=n_2+1$ 的性质,所以不存在这样的二叉树。

(6) 正确。在完全二叉树中,$n_1=0$ 或 1,而 $n_0=n_2+1$,所以分支结点个数 $n_1+n_2=n_0$ 或 $n_1+n_2=n_0-1$。

(7) 错误。完全二叉树中最多有一个单分支结点。

(8) 正确。

(9) 正确。在完全二叉树中,已知结点总数就可以确定其形态。已知其先序序列,便可知其结点总数,再根据先序序列确定每个结点的位置。实际上,只要已知完全二叉树的任何一种遍历序列就可以唯一确定该二叉树。

(10) 正确。

(11) 错误。在哈夫曼树中,权值相同的叶子结点不一定都在同一层上。例如,某个哈夫曼树中只有 3 个叶子结点,权值均为 1,它们就不可能都在同一层上。

(12) 错误。在哈夫曼树中,权值较大的叶子结点一般离根结点较近。

2. 判断以下叙述的正确性。

(1) 在一棵度为 m 的树中,每个结点最多有 $m-1$ 个兄弟。

(2) 在一棵有 n 个结点的树中,其分支数为 n。

(3) 如果树中 x 结点的层次(深度)大于 y 结点的深度,则 x 是 y 的子孙结点。

(4) 在一棵 3 次树中,有 $n_3=2,n_2=1,n_1=5$,则叶子结点个数为 6。

(5) 若一棵二叉树中的所有结点值不相同,可以由其先序序列和层次序列唯一构造出该二叉树。

(6) 若一棵二叉树中的所有结点值不相同,可以由其中序序列和层次序列唯一构造出该二叉树。

答:(1) 正确。

(2) 错误。一棵有 n 个结点的树中其分支数为 $n-1$。

(3) 错误。

(4) 正确。$m=3,n=n_0+n_1+n_2+n_3=n_0+8$,又有 $n-1=n_1+2n_2+3n_3=13,n=14$,所以 $n_0=n-8=14-8=6$。

(5) 错误。

(6) 正确。层次序列提供了根结点信息。

3. 判断以下叙述的正确性。

(1) 存在这样的二叉树,对它采用任何次序的遍历,结果相同。

(2) 二叉树就是度为 2 的树。

(3) 将一棵树转换成二叉树后,根结点没有左子树。

(4) 对于二叉树,在后序序列中,任一结点的后面都不会出现它的子孙结点。

(5) 在哈夫曼编码中,当两个字符出现的频率相同时其编码也相同。

答:(1) 正确。当二叉树只有一个根结点时,任何遍历的序列均相同。

(2) 错误。

(3) 错误。通常根结点有左子树而无右子树。

(4) 正确。

(5) 错误。哈夫曼编码是一种前缀码,即不允许出现两字符编码相同的情况。

7.3.4 简答题

1. 简述二叉树与度为 2 的树之间的差别。

答：二叉树的子树有严格的左、右之分，其次序不能任意颠倒，某个结点即使只有一棵子树，也区分是左子树还是右子树，而在度为 2 的树中，某个结点只有一棵子树时，是不区分左右性的。除此之外，二叉树可以是空树，而度为 2 的树至少有一个度为 2 的结点，所以不能为空树。

2. 已知度为 k 的树中，其度为 $1,2,\cdots,k$ 的结点数分别为 $n_1,n_2,\cdots,n_k$。求该树的结点总数 n 和叶子结点数 n_0，并给出推导过程。

答：显然有以下关系。

$$n = n_0 + n_1 + \cdots + n_k = \sum_{i=0}^{k} n_i$$

另外树中分支总数为 $n-1$，所有结点度之和等于分支总数：

$$n - 1 = n_1 + 2n_2 + \cdots + kn_k = \sum_{j=1}^{k} jn_j$$

即：

$$n = \sum_{j=1}^{k} jn_j + 1$$

所以：

$$n_0 = n - n_1 - \cdots - n_k = \sum_{j=1}^{k} jn_j - \sum_{j=1}^{k} n_j + 1 = \sum_{j=2}^{k} (j-1)n_j + 1$$

例如，若一棵度为 4 的树中度为 1、2、3、4 的结点个数分别为 4、3、2、2，求 n 和 n_0 的过程如下：

这里 $k=4$，$n_1=4$，$n_2=3$，$n_3=2$，$n_4=2$，则：

$$n = \sum_{j=1}^{k} jn_j + 1 = 4 + 2 \times 3 + 3 \times 2 + 4 \times 2 + 1 = 25$$

$$n_0 = \sum_{j=2}^{k} (j-1)n_j + 1 = 3 + 2 \times 2 + 3 \times 2 + 1 = 14$$

3. 试证明：在具有 $n(n \geqslant 1)$ 个结点的 m 次树中，若采用孩子链存储结构，则其中有 $n(m-1)+1$ 个指针域是空的。

证明：具有 n 个结点的 m 次树采用孩子链存储结构，总的结点个数为 n，每个结点有 m 个指针域，总共有 nm 个指针域。

而指向结点的非空指针域个数为 $n-1$（由指向孩子结点的分支数为 $n-1$ 个推出），所以空指针域个数 $=mn-(n-1)=n(m-1)+1$。

4. 一棵高度为 h 的完全 k 次树，如果按层次自顶向下，同一层自左向右，顺序从 1 开始对全部结点进行编号，试问：

（1）最多有多少个结点？最少有多少个结点？

（2）编号为 q 的结点的第 i 个孩子结点（若存在）编号是多少？

（3）编号为 q 的结点的双亲结点编号是多少？

答：（1）在高度为 h 的完全 k 次树中，除第 h 层以外，其余各层都是满的，即第 1 层有一

个结点，第 2 层有 k 个结点，…，第 $h-1$ 层有 k^{h-2} 个结点，第 h 层最多有 k^{h-1} 个结点（为满的情况），最少有一个结点。因此，最多结点个数是：

$$1+k+k^2+\cdots+k^{h-2}+k^{h-1}=\frac{k^h-1}{k-1}$$

最少结点个数是：

$$1+k+k^2\cdots+k^{h-2}+1=\frac{k^{h-1}-1}{k-1}+1$$

（2）设编号为 q 的结点是完全 k 次树中第 l 层上从左边数第 j 个结点，那么 q 就等于前 $l-1$ 层的结点个数加 j，即：

$$q=\frac{k^{l-1}-1}{k-1}+j$$

则

$$j=q-\frac{k^{l-1}-1}{k-1}$$

由于完全 k 次树的第 l 层上的第 j 个结点左边有 $j-1$ 个结点，它们共有 $(j-1)k$ 个孩子。因此第 j 个结点的第 i 个孩子是第 $l+1$ 层上从左边数第 $(j-1)k+i$ 个结点，其编号 p 为：

$$p=\frac{k^l-1}{k-1}+k(j-1)+i$$

将 $j=q-\frac{k^{l-1}-1}{k-1}$ 代入上式，化简得到 $p=(q-1)k+i+1$。

所以，当编号为 q 的结点存在第 i 个孩子，其编号为 $(q-1)k+i+1$。

（3）设编号为 q 的结点是完全 k 次树中第 l 层上从左边数第 j 个结点，那么：

$$j=q-\frac{k^{l-1}-1}{k-1}$$

这 j 个结点对应有 $\left\lfloor\frac{j-1}{k}\right\rfloor+1$ 个双亲结点。因此，编号为 q 的双亲结点是第 $l-1$ 层的第 $\left\lfloor\frac{j-1}{k}\right\rfloor+1$ 个结点，其编号 p 为：

$$p=\frac{k^{l-1}-1}{k-1}+\left\lfloor\frac{j-1}{k}\right\rfloor+1$$

将 $j=q-\frac{k^{l-1}-1}{k-1}$ 代入上式，化简得到 $p=\left\lfloor\frac{q+k-2}{k}\right\rfloor$。

因此，当 $q=1$ 时，该结点为根结点，无双亲结点；否则，双亲结点的编号为 $\left\lfloor\frac{q+k-2}{k}\right\rfloor$。

5. 对于如图 7.10 所示的二叉树：

（1）画出它的顺序存储结构图；

（2）将它转换（还原）成森林。

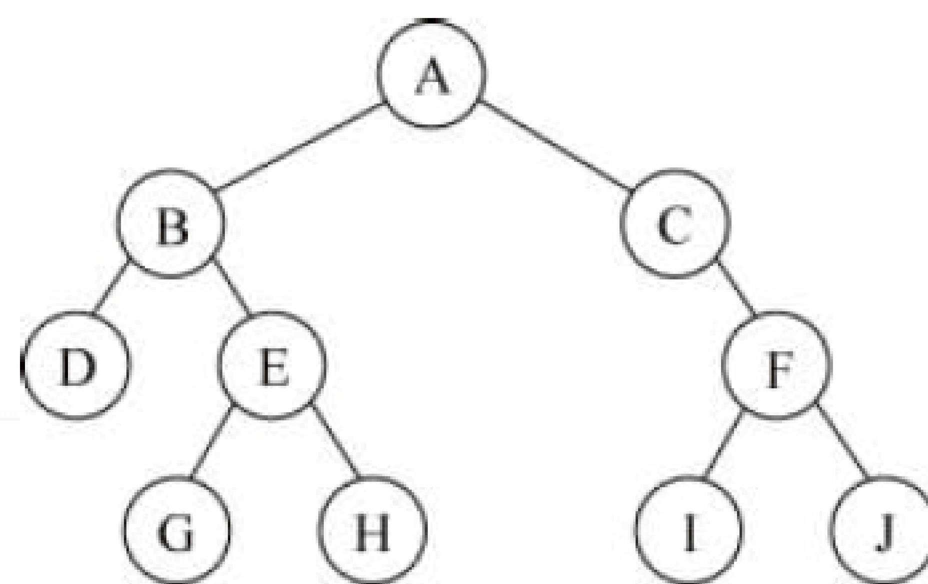


图 7.10 一棵二叉树

答：（1）它的顺序存储结构图如下：

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	B	C	D	E	#	F	#	#	G	H	#	#	I	J

(2) 转换成的森林如图 7.11 所示。

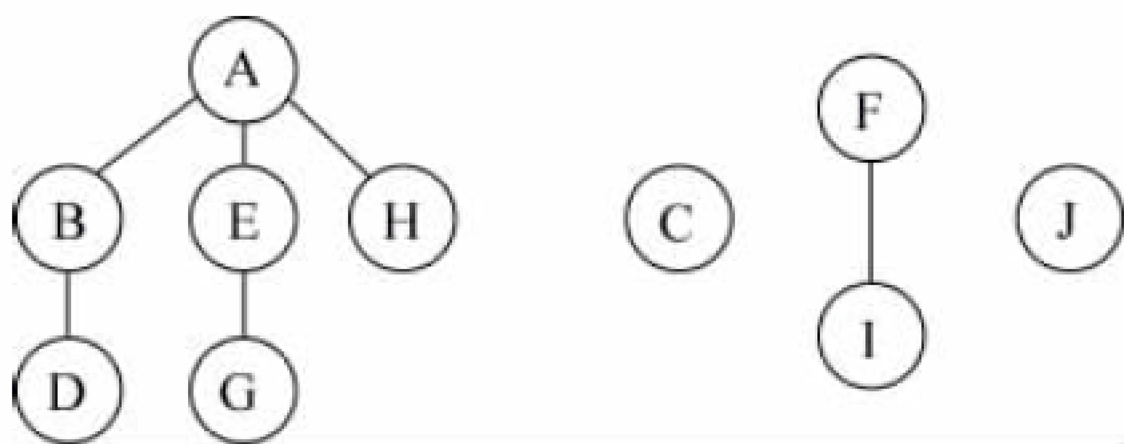


图 7.11 转换成的森林

6. 设 $F=\{T_1, T_2, T_3\}$是森林,如图 7.12 所示,试画出由 F 转换成的二叉树。

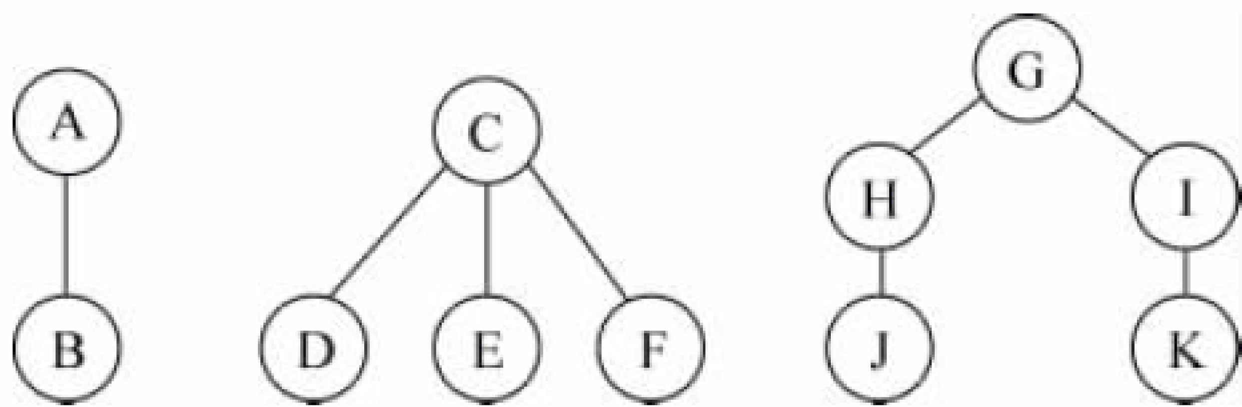


图 7.12 森林

答:由 F 转换成的二叉树如图 7.13 所示。

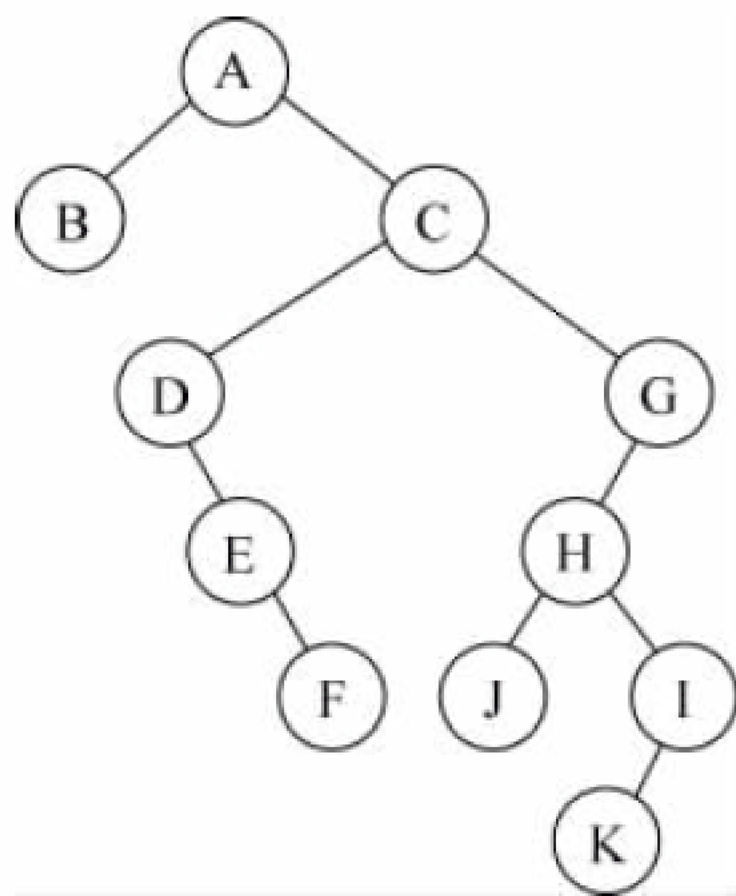


图 7.13 一棵二叉树

7. 已知一棵树 T 的先根序列与对应二叉树 B 的先序序列相同,树 T 的后根序列与对应二叉树 B 的中序序列相同。利用树的先根序列和后根序列能否唯一确定一棵树?举例说明。

答:能。由树 T 的先根序列得到二叉树 B 的先序序列,树 T 的后根序列得到 B 的中序序列,因为二叉树 B 可以由中先序序列和中序序列唯一确定,因此 B 是确定的,而二叉树 B 可以唯一还原为树 T。所以利用树的先根序列和后根序列能够唯一确定一棵树。

例如有一棵树,其先根序列为 ABCEFHIDG,后根序列为 ECHFIBDGA。构造这棵树的过程是以先序序列 ABCEFHIDG 和中序序列 ECHFIBDGA 构造一棵二叉树,如图 7.14(a)所示,然后将其还原成一棵树,如图 7.14(b)所示,该树即为所求。

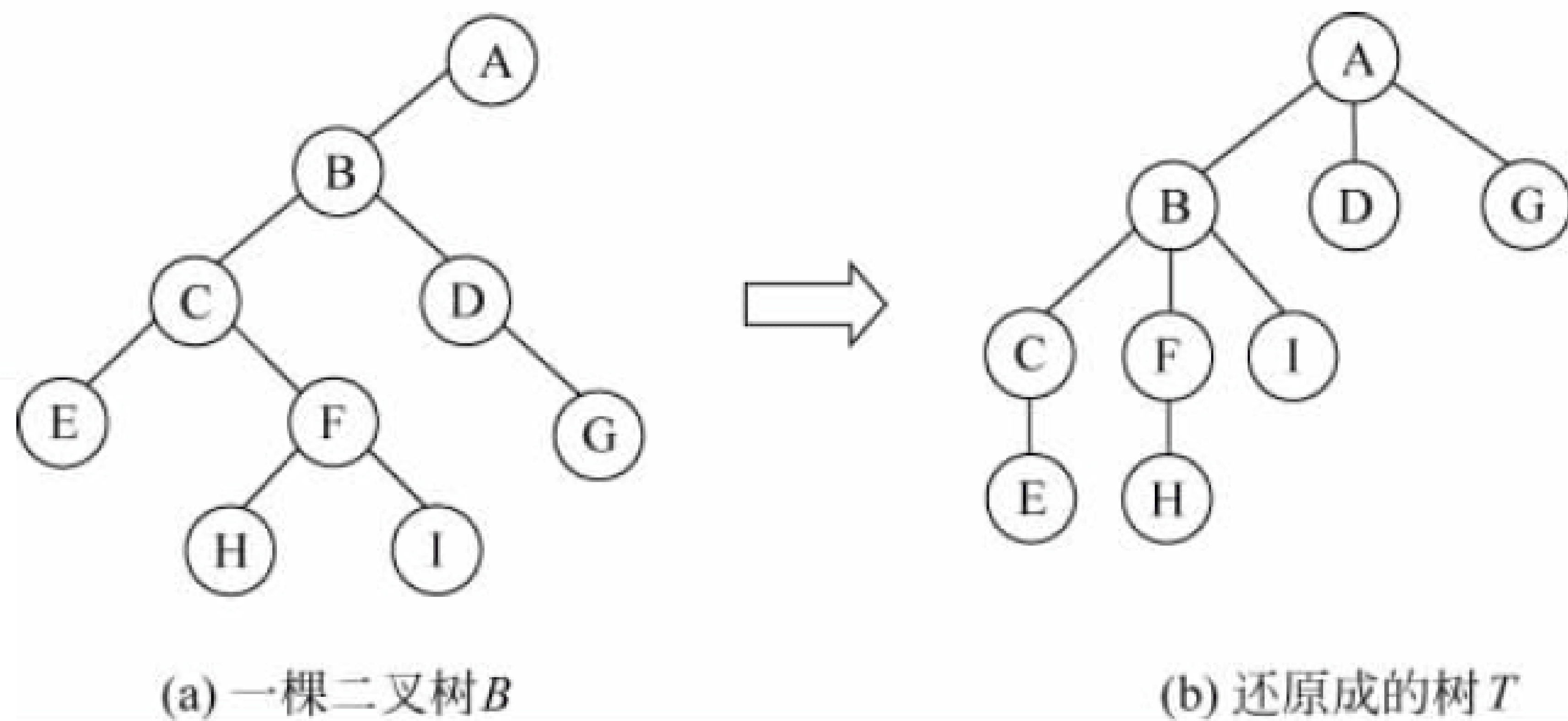


图 7.14 一棵二叉树还原成树

8. 任意一个有 n 个结点的二叉树，已知它有 m 个叶子结点，试证明非叶子结点中有 $(m-1)$ 个结点的度为2，其余度为1。

证明：设 n_1 为二叉树中度为1的结点数，n_2 为度为2的结点数，则总的结点数如下。

$$n = n_1 + n_2 + m$$

再看二叉树中的分支数，除根结点以外，其余结点都有一个分支进入，设 B 为分支数，则有：$n=B+1$。

由于这些分支由度为1和2的结点发出的，所以又有：

$$B = n_1 + 2n_2$$

由以上两式可得：

$$n=n_1+2n_2+1$$

再结合前式得：

$$n_1+n_2+m=n_1+2n_2+1$$

由此推出

$$n_2=m-1$$

9. 为什么说一棵非空完全二叉树，一旦结点个数 n 确定了，其树形也就确定了。

答：在按层序编号时，完全二叉树的结点编号为 $1\sim n$，如果已知各类结点个数，该完全二叉树的形态一定是确定的。若 n 已知，则可以根据其奇偶性确定 n_1：当 n 为偶数时，$n_1=1$，当 n 为奇数时，$n_1=0$，而 $n_0=n_2+1$，$n=n_0+n_1+n_2=2n_0-1+n_1$，$n_0=(n-n_1+1)/2$，从而 n_0 和 n_2 也确定了，所以这样的完全二叉树的形态就确定了。

10. 为什么说一棵非空完全二叉树，仅已知叶子结点个数 n_0，其树形还不能唯一确定。

答：在该完全二叉树中，n_0 已知，$n_2=n_0-1$，$n=n_0+n_1+n_2=2n_0-1+n_1$，而 n_1 可以为0或1，也就是说，$n=2n_0-1$ 或 $n=2n_0$，其结点总数不确定，所以该完全二叉树的形态是不能确定的。

11. 给定一棵非空二叉树 b，采用二叉链存储结构，说明查找中序序列的第一个结点和最后一个结点的过程。

答：中序序列的第一个结点就是根结点的最左下结点，其查找过程如下。

```
p = b;
while (p->lchild!= NULL)          //循环结束,p指向中序序列的第一个结点
    p = p->lchild;
```

中序序列的最后一个结点就是根结点的最右下结点，其查找过程如下。

```
p = b;
while (p->rchild!= NULL)          //循环结束,p指向中序序列的最后一个结点
    p = p->rchild;
```

12. 对于二叉树 T 的两个结点 a 和 b，在不构造出该二叉树的前提下，应该选择 T 的先序、中序和后序序列中的哪两个序列来判断结点 a 必定是结点 b 的祖先，并给出判断的方法。

答：可以采用先序序列和后序序列来判断。由先序序列可知结点 b 的祖先一定在 b 之前；而在后序序列中，结点 b 的祖先一定在 b 之后；取先序序列在 b 之前的结点集合以及后序序列在 b 之后的结点集合，这两个集合的交集即为 b 的祖先结点集合，则结点 a 必在该祖

先结点集合中。

13. 用一维数组存放一棵完全二叉树 ABCDEFGHIJKL，给出后序遍历该二叉树的访问结点序列。

答：该完全二叉树如图 7.15 所示，其后序序列为 HIDJKEBLFGCA。

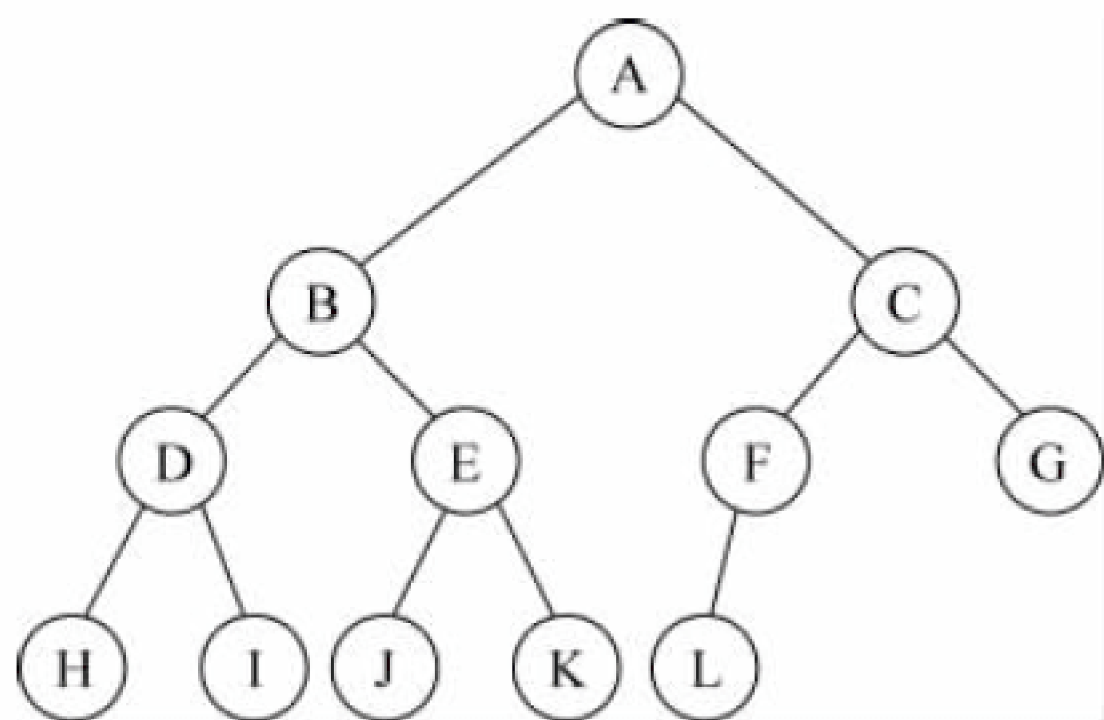


图 7.15　一棵完全二叉树

14. 已知一棵完全二叉树共有 892 个结点，试求：

(1) 树的高度；

(2) 单支结点数；

(3) 叶子结点数；

(4) 最后一个分支结点的序号。

答：(1) 该完全二叉树的高度 $h=\lceil \log_2(n+1) \rceil=\lceil \log_2 893 \rceil=10$。

(2) $n=892$ 为偶数，所以 $n_1=1$。

(3) $n_0=n_2+1$(二叉树的性质 1)，$n=n_0+n_1+n_2=2n_0$，$n_0=n/2=446$。

(4) 最后一个分支结点的序号$=\lfloor n/2 \rfloor=446$。

15. 已知完全二叉树的第 8 层有 8 个结点，则其叶子结点数是多少？

答：由完全二叉树的定义可知，除最后一层以外，其他各层的结点是满的。这里第 8 层有 8 个结点，显然第 8 层是最后的一层，那么第 7 层的结点个数为 $2^{7-1}=64$ 个，其中的 4 个结点有 8 个叶子结点，余下的为叶子结点，个数为 $64-4=60$。所以该完全二叉树的叶子结点个数$=60+8=68$ 个。

16. 若一棵二叉树的左、右子树均有 3 个结点，其左子树的先序序列与中序序列相同，右子树的中序序列与后序序列相同，试构造该树形态。

答：依题意，左子树的先序序列与中序序列相同，即有以下关系。

根左右(先序)=左根右(中序)

即以左孩子为根的子树无左孩子。

此外，右子树的中序序列与后序序列相同，即有：

左根右(中序)=左右根(后序)

即以右孩子为根的子树无右孩子。由此构造该树的形态如图 7.16 所示。

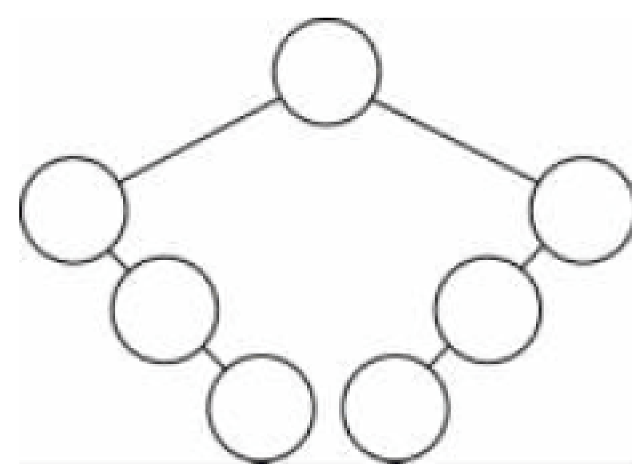


图 7.16　一棵二叉树

17. 若某非空二叉树的先序序列和中序序列正好相反，则该二叉树的形态是什么？

答：二叉树的先序序列是 NLR、中序序列是 LNR，要使 NLR=RNL（中序序列反序）成立，则 R 必须为空，所以满足条件的二叉树的形态是所有结点没有右子树的单支树。

18. 一棵二叉树的先序、中序和后序序列分别如下，其中有一部分未显示出来。试求出空格处的内容，并画出该二叉树。

先序序列：__ B __ F __ ICEH __ G

中序序列：D __ KFIA __ EJC __

后序序列：__ K __ FBHJ __ G __ A

答：由后序序列可知根结点为 A，先序序列的第一个空为 A，由中序序列可知，左子树有 5 个结点，由先序序列可知，左子树中有 B、F、I 结点，所以中序序列的第一个空为 B，可推出先序序列的第 2 个空为 D，第 3 个空为 K；右子树有 5 个结点，由中序序列可知，右子树中有 E、J、C 结点，所以先序序列中第 4 个空为 J，这样产生完整的先序序列，可知右子树根结点为 C，由中序序列可知，C 的左子树有 3 个结点，为 E、H、J，所以中序序列的第 2 个空为 H，C 的左子树只有一个结点 G，所以中序序列的第 3 个空为 G。

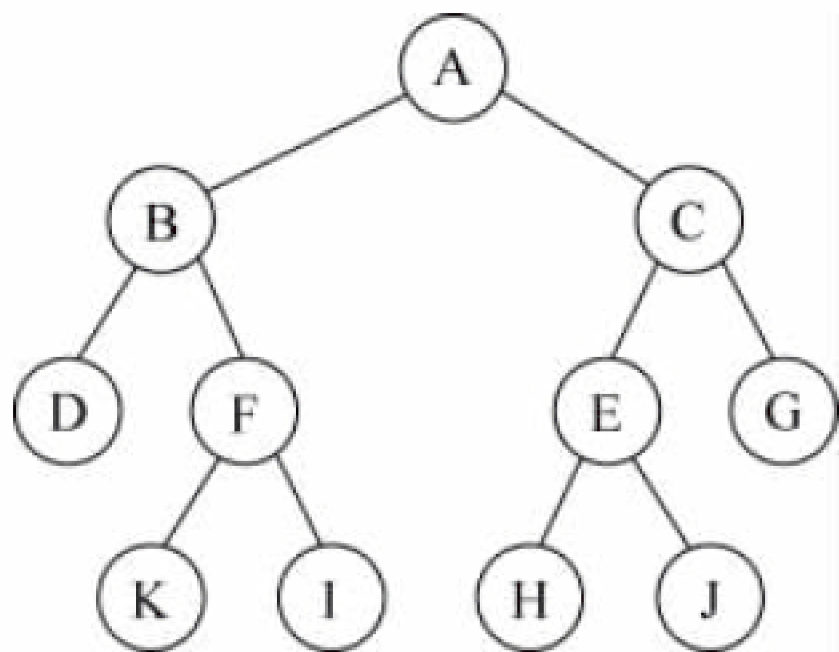


图 7.17　一棵二叉树

从而构造出的二叉树如图 7.17 所示，则先序序列为 ABDFKICEHJG；中序序列为 DBKFIAHEJCG；后序序列为 DKIFBHJEGCA。

19. 给出在中序线索二叉树 tb 中查找结点 p 的中序后继结点的过程。

答：在中序线索二叉树 tb 中，对于 p 结点有以下关系。

（1）若 p->rtag=1，表示结点 p 的右指针是线索，则 p->rchild 即为结点 p 的中序后继结点。

（2）若 p->rtag=0，表示结点 p 的右指针指向右孩子，那么它的右子树的中序遍历中的第一个结点就是结点 p 的后继结点。所以，p 结点的右孩子的最左下结点就是它的中序后继结点。

20. 一组包含不同权值的字母已经对应好哈夫曼编码，如果某个字母对应的编码为 001，则：

（1）什么编码不可能对应其他字母？

（2）什么编码肯定对应其他字母？

答：（1）由哈夫曼树的性质可知，以 0、00 和 001 为前缀的编码不可能对应其他字母。

（2）该哈夫曼树的高度至少是 3，其最少叶子结点的情况如图 7.18 所示，所以 000、01 和 1 开头的编码肯定对应其他字母。

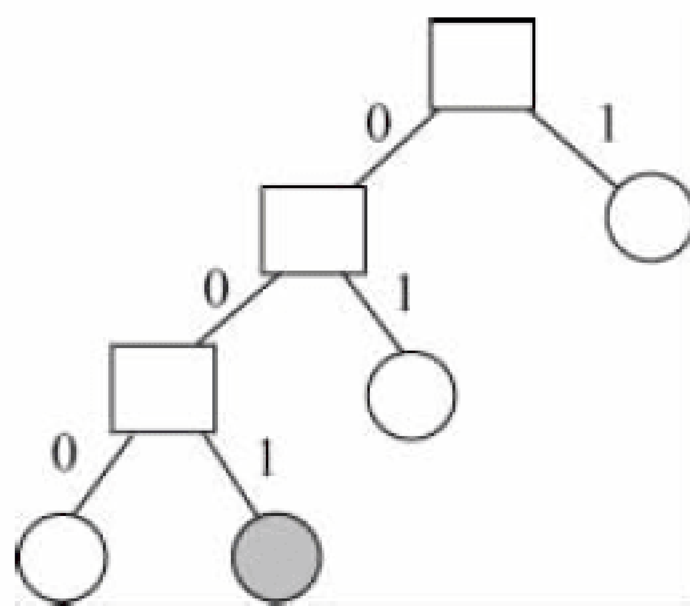


图 7.18　一棵哈夫曼树

21. 假设一段正文由字符集{a,b,c,d,e,f}中的字母构成，这 6 个字母在这段正文中出现的次数分别是{12,18,26,6,4,34}。回答以下问题：

（1）为这 6 个字母设计哈夫曼编码。

（2）求带权路径长度 WPL。

（3）设每个字节由 8 个二进制位组成，计算按照哈夫曼编码存储这段正文需要多少字节？

答：(1) 构造的哈夫曼树如图 7.19 所示，对应的哈夫曼编码如下。

a：001，b：01，c：10，d：0001，e：0000，f：11。

(2) 该哈夫曼树的 WPL=(4+6)×4+12×3+(18+26+34)×2=232。

(3) 存储这段正文所需要的二进制位数恰好等于 WPL，所以对应 232/8=29 个字节。

22. 设哈夫曼编码的长度不超过 4，若已经对两个字符编码为 1 和 01，则最多还可以对多少个字符编码？

答：在哈夫曼编码中，一个编码不能是任何其他编码的前缀。本题的哈夫曼树的最大高度为 5，而 1 和 01 已作为两个字符的编码，所以最多还有 4 个哈夫曼编码，即 0000、0001、0010 和 0011，如图 7.20 所示。

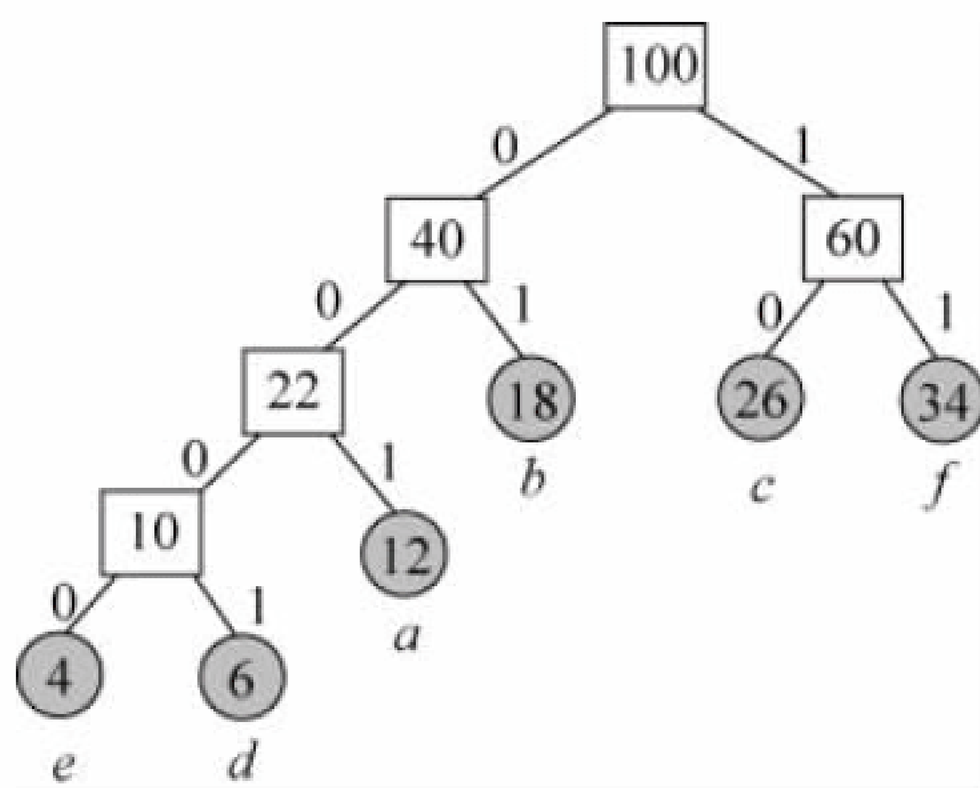


图 7.19 一棵哈夫曼树

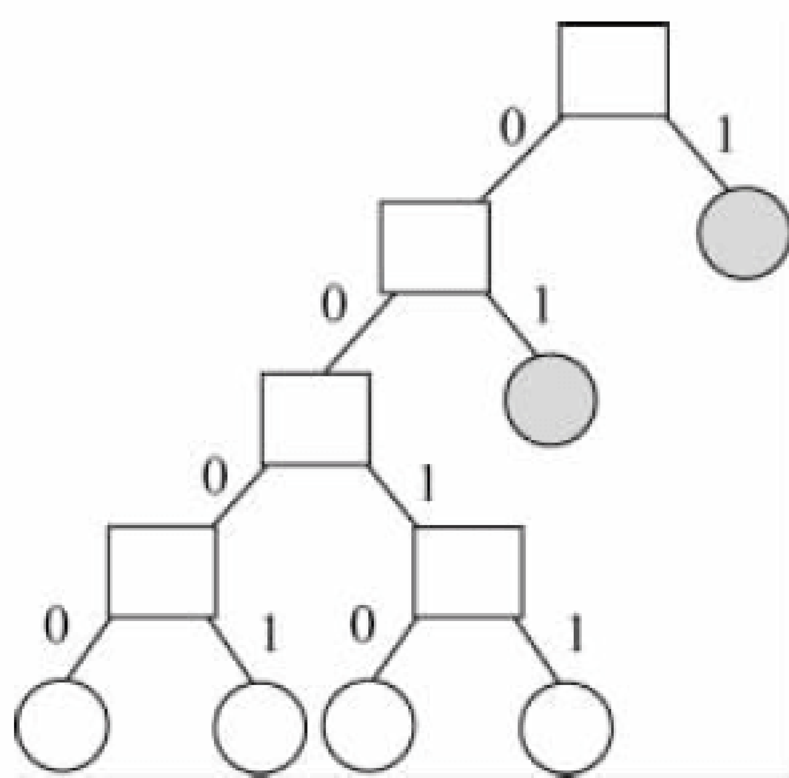


图 7.20 一棵哈夫曼树

7.3.5 算法设计题

1. 【二叉树的顺序存储结构算法】已知一棵二叉树按顺序方式存储在数组 $a[1..n]$ 中。设计一个算法，求编号分别为 i 和 j 的两个结点的最近公共祖先结点的值。

解：由二叉树顺序存储结构的特点可得到求编号 i 和 j 的两个结点的最近公共祖先结点的算法如下。

```
ElemType ancestor(SqBinTree a,int i,int j)
{   int p = i,q = j;
    while (p!= q)
        if (p > q)
            p = p/2;            //向上找 i 的祖先
        else
            q = q/2;            //向上找 j 的祖先
    return a[p];
}
```

2. 【二叉树的顺序存储结构+先序遍历算法】已知一棵含有 n 个结点的二叉树，按顺序方式存储，设计用先序遍历二叉树中结点的递归和非递归算法。

解：先序遍历的递归算法如下。

```
void PreOrder1(SqBinTree a,int i)
//a 数组存储二叉树(大小为 MaxSize),i 的初值为 1
```

```
{   if (i<MaxSize)
    {   if (a[i]!='#')
        {   printf("%c",a[i]);              //访问根结点
            PreOrder1(a,2*i);               //遍历左子树
            PreOrder1(a,2*i+1);             //遍历右子树
        }
    }
}
```

先序遍历的非递归算法如下(其思路参见《教程》7.5.3 小节先序遍历非递归算法 1):

```
void PreOrder2(SqBinTree a)                      //a 数组存储二叉树(大小为 MaxSize)
{   int St[MaxSize],top=-1,i=1;
    top++;                                       //根结点 1 进栈
    St[top]=i;
    while (top>-1)
    {   i=St[top];top--;                         //出栈结点 i
        printf("%c",a[i]);
        if (2*i+1<MaxSize && a[2*i+1]!='#')      //右孩子进栈
        {   top++;
            St[top]=2*i+1;
        }
        if (2*i<MaxSize && a[2*i]!='#')          //左孩子进栈
        {   top++;
            St[top]=2*i;
        }
    }
}
```

3.【二叉链存储结构+先序递归遍历算法】假设二叉树中每个结点值为单个字符,采用二叉链存储结构存储。设计一个算法计算一棵给定二叉树 b 中的所有双分支结点个数。

解:采用基于先序遍历的递归方法。对应的算法如下:

```
int DSonNodes(BTNode *b)
{   int num1,num2,n;
    if (b==NULL)
        return 0;
    else if (b->lchild!=NULL && b->rchild!=NULL)
        n=1;                          //为双分支结点
    else
        n=0;                          //其他
    num1=DSonNodes(b->lchild);        //递归求左子树的双分支结点数
    num2=DSonNodes(b->rchild);        //递归求右子树的双分支结点数
    return (num1+num2+n);
}
```

4.【二叉链存储结构+先序递归遍历算法】假设二叉树中每个结点值为单个字符(所有结点值不相同),采用二叉链存储结构存储。现有一个算法 DestroyBTree(b)用于删除并

释放以 b 为根结点的子树，要求设计一个算法利用它删除二叉树 b 中以结点值 x 为根结点的子树。

解：采用基于先序遍历的递归方法，首先查找值 x 为结点 p，然后调用 DestroyBTree(p)删除并释放该子树。对应的算法如下：

```
void Delx(BTNode *&b,ElemType x)
{   if (b!= NULL)
    {   if (b->data == x)
        {   DestroyBTree(b);            //调用二叉树基本运算 DestroyBTree 算法
            b = NULL;
        }
        else
        {   Delx(b->lchild,x);
            Delx(b->rchild,x);
        }
    }
}
```

5.【二叉链存储结构+先序递归遍历算法】假设二叉树中每个结点值为单个字符(所有结点值不相同)，采用二叉链存储结构存储。设计一个算法 void findparent(BTNode *b, char x,BTNode *&p)求二叉树 b 中指定值为 x 的结点的双亲结点 p。提示：根结点的双亲为 NULL，若在 b 中未找到值为 x 的结点，p 也为 NULL，并假设二叉树中所有结点值是唯一的。

解：采用基于先序遍历的递归方法，先判断根结点，若不满足要求，再到左子树中查找，若没有找到，最后到右子树中查找。对应的算法如下：

```
void Findparent(BTNode *b,char x,BTNode *&p)
{   if (b!= NULL)
    {   if (b->data == x) p = NULL;
        else if (b->lchild!= NULL && b->lchild->data == x)
            p = b;
        else if (b->rchild!= NULL && b->rchild->data == x)
            p = b;
        else
        {   Findparent(b->lchild,x,p);
            if (p == NULL)
                Findparent(b->rchild,x,p);
        }
    }
    else p = NULL;
}
```

6.【二叉链存储结构+先序递归遍历算法】假设二叉树中每个结点值为单个字符，采用二叉链存储结构存储。设计一个算法，求该二叉树中距离根结点最近的叶子结点。

解：采用基于先序遍历的递归方法，用 min 记录结点层次(初始值取一个大整数)，l 表示当前访问结点的层次。先判断根结点，若为叶子结点，其层次 l 小于 min，置 min=l，用 x

记录其结点值；再到左子树中查找并比较，最后到右子树中查找并比较。对应的算法如下：

```
void MinlenLeaf(BTNode *b,int l,int &min,char &x)
{   //l的初值为1,min取最大整数,x为所求的叶子结点
    if (b!=NULL)
    {   if (b->lchild==NULL && b->rchild==NULL)
        {   if (l<min)
            {   min=l;
                x=b->data;
            }
        }
        MinlenLeaf(b->lchild,l+1,min,x);
        MinlenLeaf(b->rchild,l+1,min,x);
    }
}
```

7.【二叉链存储结构+先序递归遍历算法】假设二叉树采用二叉链存储结构，设计一个算法判断一棵二叉树是否为对称同构。所谓对称同构是指二叉树中任何结点的左、右子树结构是相同的。

解：采用基于先序遍历的递归方法，先判断根结点的左、右孩子是否对称同构，若不是返回假。再判断左子树，最后判断右子树。对应的算法如下：

```
bool isomorphism(BTNode *b)
{   if (b==NULL) return true;
    if ((b->lchild==NULL && b->rchild!=NULL) ||
        (b->lchild!=NULL && b->rchild==NULL))
        return false;
    return isomorphism(b->lchild) & isomorphism(b->rchild);
}
```

8.【二叉链存储结构+先序递归遍历算法】假设二叉树采用二叉链存储结构存储，试设计一个算法，输出从每个叶子结点到根结点的逆路径。

解：采用基于先序遍历的递归方法，用 path 数组存放查找的路径，pathlen 存放路径长度，当找到叶子结点 b 时，由于 b 叶子结点尚未添加到 path 中，因此在输出路径时还需输出 b—> data 值；若 b 不为叶子结点，将 b—> data 放入 path 中，然后在左、右子树中递归查找。递归算法如下：

```
void AllPath1(BTNode *b,ElemType path[],int pathlen)
{   //b为根结点时,pathlen初始值为0
    int i;
    if (b!=NULL)
    {   if (b->lchild==NULL && b->rchild==NULL)        //b为叶子结点
        {   printf("  %c到根结点逆路径: %c ",b->data,b->data);
            for (i=pathlen-1;i>=0;i--)
                printf("%c ",path[i]);
            printf("\n");
        }
```

```
        else
        {   path[pathlen] = b->data;            //将当前结点放入路径中
            pathlen++;                          //路径长度增1
            AllPath1(b->lchild,path,pathlen);   //递归扫描左子树
            AllPath1(b->rchild,path,pathlen);   //递归扫描右子树
            pathlen--;                          //恢复环境
        }
    }
}
```

9. 【二叉链存储结构+先序递归遍历算法】假设二叉树采用二叉链存储结构存储，设计一个算法，输出该二叉树中第一条最长的路径长度，并输出此路径上各结点的值。

解：采用基于先序递归算法的思路。用 path 保存扫描到当前结点的路径，pathlen 保存扫描到当前结点的路径长度，longpath 保存最长的路径，longpathlen 保存最长路径长度。当 b 为空时，表示当前扫描的一个分支已扫描完毕，将 pathlen 与 longpathlen 进行比较，将较长的路径及路径长度分别保存在 longpath 和 longpathlen 中。对应的算法如下：

```
#include "btree.cpp"                        //二叉树基本运算算法
void LongPath(BTNode *b,ElemType path[],int pathlen,ElemType longpath[],
    int &longpathlen)                       //pathlen 和 longpathlen 的初值为 0
{   if (b==NULL)
    {   if (pathlen>longpathlen)            //若当前路径更长，将路径保存在 longpath 中
        {   for (int i=pathlen-1;i>=0;i--)
                longpath[i]=path[i];
            longpathlen=pathlen;
        }
    }
    else
    {   path[pathlen]=b->data;              //将当前结点放入路径中
        pathlen++;                          //路径长度增1
        LongPath(b->lchild,path,pathlen,longpath,longpathlen);
                                            //递归扫描左子树
        LongPath(b->rchild,path,pathlen,longpath,longpathlen);
                                            //递归扫描右子树
    }
}
```

设计以下主函数：

```
int main()
{   BTNode *b;
    CreateBTree(b,"A(B(D,E(G,H)),C(,F(I)))");     //《教程》中图 7.11 所示的二叉树
    printf("b:"); DispBTree(b); printf("\n");
    ElemType path[MaxSize],longpath[MaxSize];
    int longpathlen=0;
    LongPath(b,path,0,longpath,longpathlen);
    printf("第一条最长逆路径长度：%d\n",longpathlen);
```

```
    printf("第一条最长逆路径:");
    for (int i = longpathlen - 1;i >= 0;i -- )
        printf(" %c ",longpath[i]);
    printf("\n");
    DestroyBTree(b);
    return 1;
}
```

程序的执行结果如下：

```
括号表示法: A(B(D,E(G,H)),C(,F(I)))
第一条最长逆路径长度:4
第一条最长逆路径: G E B A
```

10.【二叉链存储结构＋先序递归遍历算法】假设二叉树采用二叉链存储结构进行存储，设计一个算法，采用先序遍历方法求二叉树 b 的宽度(即具有结点数最多的那一层上的结点总数)。

解：采用基于先序递归算法的思路，首先置数组 width 的所有元素为 0，当先序遍历到某个结点 b 时，求出其层次为 l，将 width[l]增 1。遍历完毕，比较求出 width 中的最大元素值即为二叉树 b 的宽度。对应的算法如下：

```
void Width1(BTNode *b,int l,int width[])
{   //l 的初值为 1
    if (b!= NULL)
    {   width[l]++;
        Width1(b->lchild,l+1,width);
        Width1(b->rchild,l+1,width);
    }
}
int Width(BTNode *b)          //求二叉树 b 的宽度
{   int i,max = 0;
    int width[MaxSize];
    for (i = 1;i < MaxSize;i++)
        width[i] = 0;
    Width1(b,1,width);
    for (i = 1;i < MaxSize;i++)
        if (width[i]> max) max = width[i];
    return max;
}
```

11.【二叉链存储结构＋先序递归遍历算法】假设二叉树的存储结构如下：

```
typedef struct node
{   ElemType data;
    struct node *lchild, *rchild;
    struct node *parent;              //双亲指针
} PBTNode;
```

其中,结点的 lchild 和 rchild 已分别填有指向左、右孩子结点的指针,而 parent 域中为空(拟作为指向双亲结点的指针)。设计一个算法,将该存储结构中各结点的 parent 域的值修改成指向其双亲结点的指针。

解:采用先序遍历的递归算法求解,*p* 指向当前访问结点 *b* 的双亲,初始时,结点 *b* 为根结点,*p* 为 NULL。对应的算法如下:

```
void setparent(PBTNode * b,PBTNode * p)          //p 的初始值为 NULL
{   if (b!= NULL)
    {   b->parent = p;
        setparent(b->lchild,b);
        setparent(b->rchild,b);
    }
}
```

12.【二叉链存储结构+先序递归遍历算法】假设二叉树采用二叉链存储结构存储。设计一个算法,利用结点的右孩子指针 rchild 将一棵二叉树的叶子结点按从左往右的顺序串成一个单链表。

解:采用先序遍历的递归算法求解,head 是建立的单链表首结点指针(初始时为空),tail 是尾结点指针。当先序遍历到结点 *b* 时,若它是叶子结点,如果 head 为空表示该结点是遇到的第一个叶子结点,设置 head=tail=*b*;否则它不是第一个叶子结点,将其链到 tail 结点之后。再遍历左右子树。对应的算法如下:

```
void Link(BTNode * b,BTNode * &head,BTNode * &tail)
{   if (b!= NULL)
    {   if (b->lchild == NULL && b->rchild == NULL)      //叶子结点
        {   if (head == NULL)                            //第一个叶子结点
            {   head = b;
                tail = b;
            }
            else                                         //其他叶子结点
            {   tail->rchild = b;
                tail = b;
            }
        }
        Link(b->lchild,head,tail);
        Link(b->rchild,head,tail);
    }
}
```

13.【二叉链存储结构+先序递归遍历算法】假设二叉树中每个结点值为单个字符,采用二叉链存储结构存储。设计一个算法求二叉树 *b* 的最小枝长。所谓最小枝长是指根结点到最近叶子结点的路径长度。

解:采用基于先序遍历的递归方法,先判断根结点不空,再求出左、右子树的最小枝长 min1 和 min2,并返回 MIN(min1,min2)+1。对应的算法如下:

```
int MinBranch(BTNode * b)
{   int min1,min2,min;
    if (b == NULL)
        return 0;
    else
    {   min1 = MinBranch(b -> lchild);
        min2 = MinBranch(b -> rchild);
        if (min1 < min2) min = min1 + 1;
        else min = min2 + 1;
        return min;
    }
}
```

14.【二叉链存储结构＋先序递归遍历算法】假设二叉树中每个结点值为单个字符，采用二叉链存储结构存储。设计一个算法，求二叉树 b 中第 k 层上的叶子结点个数。

解：采用基于先序遍历的递归方法，num 置初值 0，若当前访问结点 b 是第 k 层上的叶子结点，则 num＋＋，再求出左、右子树第 k 层上的叶子结点个数 min1 和 min2，最后返回 min1＋min2。对应的算法如下：

```
int LevelLeafkCount(BTNode * b,int h,int k)
{   //h 的初值为 1
    int num1,num2,num = 0;
    if (b!= NULL)
    {   if (h == k && b -> lchild == NULL && b -> rchild == NULL)
            num++;
        num1 = LevelLeafkCount(b -> lchild,h + 1,k);
        num2 = LevelLeafkCount(b -> rchild,h + 1,k);
        num += num1 + num2;
        return num;
    }
    else return 0;
}
int LevelLeafk(BTNode * b,int k)            //求 b 中第 k 层上的叶子结点个数
{
    return LevelLeafkCount(b,1,k);
}
```

15.【二叉链存储结构＋后序遍历算法】假设二叉树采用二叉链存储结构存储，要求返回二叉树 b 的后序序列中的第一个结点的指针，是否可以不用递归且不用栈来完成？请简述原因。

解：可以。二叉树后序序列中的第一个结点即是左子树中最左下的结点，若最左下的结点无左子树但有右子树，那么后序序列第一个结点应是该右子树中最左下的结点，依此类推。对应的算法如下：

```
BTNode * postfirst(BTNode * b)
{   BTNode * p = b;
```

```
        if (b!= NULL)
            while (p->lchild!= NULL || p->rchild!= NULL)
            {   while (p->lchild!= NULL)           //先找到结点 p 的最左下结点
                    p = p->lchild;
                if (p->rchild!= NULL)              //若结点 p 有右孩子,转向该右孩子
                    p = p->rchild;
            }
        return p;                                  //找到的第一个叶子结点 p 即为所求
    }
```

16.【二叉链存储结构+后序递归遍历算法】假设二叉树中每个结点值为单个字符,采用二叉链存储结构存储。试设计一个算法,采用后序遍历方式求一棵给定二叉树 b 中的所有小于 x 的结点个数。

解:对应的算法如下。

```
int LessNodes(BTNode *b, char x)
{   int num1, num2, num = 0;
    if (b == NULL)
        return 0;
    else
    {   num1 = LessNodes(b->lchild, x);
        num2 = LessNodes(b->rchild, x);
        num += num1 + num2;
        if (b->data < x) num++;
        return num;
    }
}
```

17.【二叉链存储结构+后序递归遍历算法】假设二叉树中每个结点值为单个字符,采用二叉链存储结构存储。设计一个算法,求一个二叉树 b 中的最大结点值,空树返回'0'。

解:求一个二叉树中的最大结点值的递归模型如下。

$$f(b)=\begin{cases}\text{'0'} & \text{当 } b=\text{NULL} \text{ 时}\\ b\text{—>data} & \text{当 } b \text{ 只有一个结点时}\\ \text{MAX}\{f(b\text{—>lchild}),f(b\text{—>rchild}),b\text{—>data}\} & \text{其他情况}\end{cases}$$

对应的基于后序遍历的算法如下:

```
ElemType maxnode(BTNode *b)
{   ElemType max = b->data, max1;
    if (b!= NULL)
    {   if (b->lchild == NULL && b->rchild == NULL)       //只有一个结点时
            return b->data;
        else
        {   if (b->data > max)
                max = b->data;
            if (b->lchild!= NULL)
```

```
                max1 = maxnode(b -> lchild);          //遍历左子树
            if (max1 > max) max = max1;
            if (b -> rchild!= NULL)
                max1 = maxnode(b -> rchild);          //遍历右子树
            if (max1 > max) max = max1;               //求最大值
            return max;                               //返回最大值
        }
    }
    return '0';
}
```

18.【二叉链存储结构+后序递归遍历算法】假设一个仅包含二元运算符的简单算术表达式以二叉链形式存储在二叉树 b 中,写出计算该算术表达式值的算法。

解:以二叉树表示算术表达式,根结点用于存储运算符。若能先分别求出左子树和右子树表示的子表达式的值,最后就可以根据根结点的运算符的要求计算出表达式的最后结果。对应的算法如下:

```
typedef struct node
{   double val;                        //存放值
    char optr;                         //只取'+'、'-'、'*'、'/'
    struct node *lchild, *rchild;
} BTNode;
double compval(BTNode *b)              //用后序遍历方法求二叉树表示的算术表达式的值
{   double lv,rv,value;
    if (b!= NULL)
    {   if (b -> lchild == NULL && b -> rchild == NULL)
            return b -> val;           //为叶子结点时返回其值
        else
        {   lv = compval(b -> lchild);  //求左子树表示的子表达式的值
            rv = compval(b -> rchild);  //求右子树表示的子表达式的值
            switch(b -> optr)
            {
            case '+':value = lv + rv;
                    break;
            case '-':value = lv - rv;
                    break;
            case '*':value = lv * rv;
                    break;
            case '/':if (rv!= 0) value = lv/rv;
                     else exit(0);
                    break;
            }
            return value;
        }
    }
    else return 0;
}
```

19.【二叉链存储结构+后序非递归遍历算法】假设二叉树中每个结点值为单个字符，采用二叉链存储结构存储。b 指向根结点，其中有两个结点值分别为 x、y 的结点(假设它们不互为祖先)。设计一个算法求出结点 x 和 y 的最近的共同祖先。

解：本题采用非递归后序遍历算法。不失一般性，假设 x 结点在 y 结点的左边。当后序遍历访问到 x 结点时，此时栈 St 中的所有结点均为 x 结点的祖先，此时将其复制到 anor 数组中，然后继续后序遍历访问到 y 结点，同样此时栈 St 中的所有结点均为 y 结点的祖先，再将其与 anor 中的结点依次(从 0 开始)比较，找出最近的共同祖先。对应的算法如下：

```
bool ancestor(BTNode *b,char x,char y)
{   BTNode *St[MaxSize],*p;
    int i,top=-1;                        //栈顶指针置初值
    bool flag;
    ElemType anor[MaxSize];
    do
    {   while (b)                        //将b的所有左结点进栈
        {   top++;
            St[top]=b;
            b=b->lchild;
        }
        p=NULL;                          //p指向当前结点的前一个已访问的结点
        flag=true;                       //flag为真表示正在处理栈顶结点
        while (top!=-1 && flag)
        {
            b=St[top];                   //取出当前的栈顶元素
            if (b->rchild==p)            //右子树不存在或已被访问,访问之
            {   if (b->data==x)          //要访问的结点为要找的结点x
                {   for (i=0;i<=top;i++) //将路径存入anor中
                        anor[i]=St[i]->data;
                    top--;
                    p=b;                 //p指向刚访问过的结点
                }
                else if (b->data==y)     //要访问的结点为要找的结点y
                {   i=0;
                    while (anor[i]==St[i]->data)     //查找最近公共祖先
                        i++;
                    printf("最近公共祖先:%c\n",anor[i-1]);
                    return true;
                }
                else
                {   top--;
                    p=b;                 //p指向则被访问的结点
                }
            }
            else
            {   b=b->rchild;             //b指向右子树
                flag=false;              //表示当前不是处理栈顶结点
```

```
            }
        }
    } while (top!= -1);
    return false;
}
```

也可以采用这样的遍历方法：设 $f(b,x,y)$ 返回找到的最近的共同祖先的指针。如果当前结点 b 是 x 或者 y 结点之一，它就是最近的共同祖先，返回 b；否则，递归对左、右子树查找，如果左、右子树返回的最近的共同祖先均不为空，说明 x、y 结点分别在当前结点左、右两边，则返回当前结点 b，若一个不为空，返回不为空的结果，若都为空，返回空。对应的算法如下：

```
BTNode *ancestor1(BTNode *b,char x,char y)
{   BTNode *p,*q;
    if (b==NULL) return NULL;
    if (b->data==x || b->data==y)
        return b;
    p=ancestor1(b->lchild,x,y);
    q=ancestor1(b->rchild,x,y);
    if (p!=NULL && q!=NULL) return b;
    if (p!=NULL) return p;
    if (q!=NULL) return q;
    return NULL;
}
```

20.【二叉链存储结构+层次遍历算法】假设二叉树中每个结点值为单个字符，采用二叉链存储结构存储。设计一个算法，采用层次遍历的方法求二叉树 b 的宽度(即具有结点数最多的那一层上的结点总数)。

解：采用层次遍历的方法求出所有结点的层编号，然后求出各层的结点总数，通过比较找出层结点总数最多的值。对应的算法如下：

```
#include "btree.cpp"                        //二叉树基本运算算法
int BTWidth(BTNode *b)
{   struct
    {   int lno;                            //结点的层次编号
        BTNode *p;                          //结点指针
    } Qu[MaxSize];                          //定义顺序非循环队列
    int front,rear;                         //定义队首和队尾指针
    int lnum,max,i,n;
    front=rear=0;                           //置队列为空队
    if (b!=NULL)
    {   rear++;
        Qu[rear].p=b;                       //根结点进队
        Qu[rear].lno=1;                     //根结点的层次编号为1
        while (rear!=front)                 //队不空循环
```

```
            {   front++;
                b = Qu[front].p;                    //出队 b
                lnum = Qu[front].lno;               //求其层次
                if (b -> lchild!= NULL)             //左孩子进队
                {   rear++;
                    Qu[rear].p = b -> lchild;
                    Qu[rear].lno = lnum + 1;        //孩子的层次为 lumn + 1
                }
                if (b -> rchild!= NULL)             //右孩子进队
                {   rear++;
                    Qu[rear].p = b -> rchild;
                    Qu[rear].lno = lnum + 1;        //孩子的层次为 lumn + 1
                }
            }
            printf("各结点的层编号:\n");            //输出各结点的层编号
            for (i = 1;i <= rear;i++)
                printf("\t%c, %d\n",Qu[i].p -> data,Qu[i].lno);
            max = 0;lnum = 1;i = 1;
            while (i <= rear)                       //通过队列求二叉树 b 的宽度
            {   n = 0;
                while (i <= rear && Qu[i].lno == lnum)
                {   n++;
                    i++;
                }
                lnum = Qu[i].lno;
                if (n > max) max = n;
            }
            return max;
        }
        else
            return 0;
}
```

设计以下主函数：

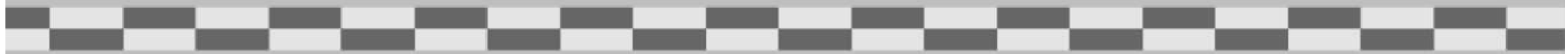
```
int main()
{   BTNode  *b;
    CreateBTree(b,"A(B(D,E(G,H)),C(,F(I)))");       //《教程》中图 7.11 所示的二叉树
    printf("b:"); DispBTree(b); printf("\n");
    printf("二叉树的宽度: %d\n",BTWidth(b));
    DestroyBTree(b);
    return 1;
}
```

程序的执行结果如下：

```
b: A(B(D,E(G,H)),C(,F(I)))
各结点的层编号:
```

```
        A,1
        B,2
        C,2
        D,3
        E,3
        F,3
        G,4
        H:4
        I:4
    二叉树的宽度:3
```

第8章 图



8.1 本章知识体系

1. 知识结构图

本章的知识结构如图 8.1 所示。



图 8.1　第 8 章知识结构图

2. 基本知识点

(1) 图的定义和相关术语。

(2) 图的邻接矩阵和邻接表两种主要存储结构及其特点。

(3) 图的基本运算算法设计。

(4) 图的深度优先和广度优先遍历算法。

(5) 图的两种遍历算法在图搜索算法设计中的应用。

(6) 生成树和最小生成树的定义，求最小生成树的 Prim 和 Kruskal 算法。

(7) 求单源最短路径的 Dijkstra 算法，求多源最短路径的 Flody 算法。

(8) 拓扑排序过程。

(9) 求 AOE 网关键路径的过程。

(10) 灵活地运用图这种数据结构解决一些综合应用问题。

3. 要点归纳

(1) 图由两个集合组成，$G=(V,E)$，V 是顶点的有限集合，E 是边的有限集合。

(2) 在有向图 $G=(V,E)$ 中，集合 E 中的元素为有序对。

(3) 在无向图 $G=(V,E)$ 中，集合 E 中的元素为无序对。无向图可以看成有向图的特殊情况。

(4) 如果图中从顶点 u 到顶点 v 之间存在一条路径，则称 u 和 v 是连通的。

(5) 如果无向图 G 中任意两个顶点都是连通的，称 G 为连通图。无向图 G 的极大连通子图称为 G 的连通分量。

(6) 有向图 G 中任意两个顶点都是连通的，称 G 为强连通图。有向图 G 的极大强连通子图称为 G 的强连通分量。

(7) 图的主要存储结构有邻接矩阵和邻接表。

(8) 无向图的邻接矩阵一定是对称矩阵，但对称矩阵对应的图不一定都是无向图。

(9) 一个图的邻接矩阵是不对称的，则该图一定是有向图。

(10) 若用邻接表表示图，图中的每个顶点 v 都对应一个单链表。单链表 v 中每个结点存放的顶点 u 满足 $\langle v,u\rangle \in E(G)$。

(11) 对于连通图，从它的任一顶点出发进行一次深度优先遍历或深度优先遍历可访问到图的每个顶点。

(12) 对于非连通图，它有几个连通分量就需要调用几次深度优先遍历或深度优先遍历才能访问图的全部顶点。

(13) 图的深度优先遍历与二叉树的先序遍历类似。

(14) 图的广度优先遍历与二叉树的层次遍历类似。

(15) 给定一个不带权的连通图，采用深度优先遍历可以找到从顶点 v 到 u 的所有路径，而采用广度优先遍历可以找到最短路径。

(16) 如果树 T 是图 G 的一个子图，且 $V(T)=V(G)$，即 G 的所有顶点也都是 T 的顶点，则称 T 为 G 的生成树。

(17) 一个带权无向图的最小生成树并非指边数最少的生成树(因为所有生成树的边数相同)，而是指所有边权值之和最小的生成树。

(18) 一个带权无向图的最小生成树不一定是唯一的，但最小生成树的所有边权值之和一定是唯一的。

(19) 一个图的最短路径一定是简单路径。

(20) 求单源最短路径的 Dijkstra 算法既适合于带权有向图也适合于带权无向图。

(21) 在 Dijkstra 算法中一旦考查了一个顶点(把它添加到 S 集合中)，它以后的最短路径不会再调整。

(22) Dijkstra 算法不适合含负权值的图求单源最短路径。

(23) Floyd 算法可以对含负权值的图求最短路径，但图中不能有权值和为负数的环路。

(24) 一个有向图中如果存在回路，则不能产生完整的拓扑序列，所以一个强连通图是不能进行拓扑排序的。

(25) 若一个有向图不能产生完整的拓扑序列，则其中必存在回路。

(26) 如果一个有向图的拓扑序列是唯一的，则图中必定仅有一个顶点的入度为 0，一个顶点的出度为 0。

(27) 一个 AOE 网中至少有一条关键路径，且是从源点到汇点的路径中最长的一条。

(28) 一个 AOE 网的关键路径不一定是唯一的,但其关键路径长度一定是唯一的。

8.2 教材中的练习题及参考答案

1. 图 G 是一个非连通图,共有 28 条边,则该图最少有多少个顶点?

答:由于 G 是一个非连通图,在边数固定时,顶点数最少的情况是该图由两个连通分量构成,且其中之一只含一个顶点(没有边),另一个为完全无向图。设该完全无向图的顶点数为 n,其边数为 $n(n-1)/2$,即 $n(n-1)/2=28$,得 $n=8$。所以,这样的非连通图最少有 $1+8=9$ 个顶点。

2. 有一个如图 8.2(a)所示的有向图,给出其所有的强连通分量。

答:图中顶点 0、1、2 构成一个环,这个环一定是某个强连通分量的一部分。再考查顶点 3、4,它们到这个环中的顶点都有双向路径,所以将顶点 3、4 加入。考查顶点 5、6,它们各自构成一个强连通分量。该有向图的强连通分量有 3 个,如图 8.2(b)所示。

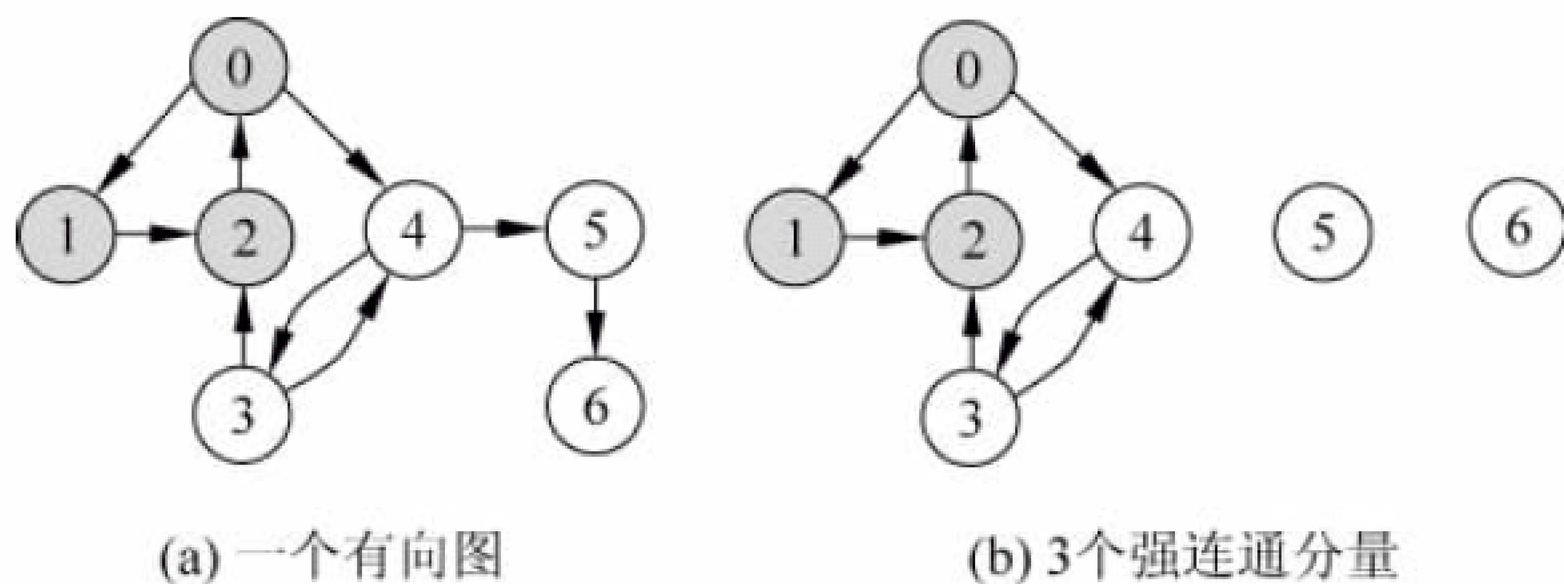


图 8.2 一个有向图及其强连通分量

3. 对于稠密图和稀疏图,采用邻接矩阵和邻接表哪个更好一些?

答:邻接矩阵适合于稠密图,因为邻接矩阵占用的存储空间与边数无关。邻接表适合于稀疏图,因为邻接表占用的存储空间与边数有关。

4. 对于 n 个顶点的无向图和有向图(均为不带权图),将采用邻接矩阵和邻接表表示时如何求解以下问题:

(1) 图中有多少条边?

(2) 任意两个顶点 i 和 j 是否有边相连?

(3) 任意一个顶点的度是多少?

答:(1) 对于邻接矩阵表示的无向图,图的边数等于邻接矩阵数组中为 1 的元素个数除以 2;对于邻接表表示的无向图,图中的边数等于边结点的个数除以 2。

对于邻接矩阵表示的有向图,图中的边数等于邻接矩阵数组中为 1 的元素个数;对于邻接表表示的有向图,图中的边数等于边结点的个数。

(2) 对于邻接矩阵 g 表示的无向图,邻接矩阵数组元素 g.edges[i][j]为 1 表示它们有边相连,否则为无边相连。对于邻接矩阵 g 表示的有向图,邻接矩阵数组元素 g.edges[i][j]为 1 表示从顶点 i 到顶点 j 有边,g.edges[j][i]为 1 表示从顶点 j 到顶点 i 有边。

对于邻接表 G 表示的无向图,若从头结点 G->adjlist[i]的单链表中找到编号为 j 的边表结点,表示它们有边相连,否则为无边相连。对于邻接表 G 表示的有向图,若从头结点

G->adjlist[i]的单链表中找到编号为 j 的边表结点，表示从顶点 i 到顶点 j 有边。若从头结点 G->adjlist[j]的单链表中找到编号为 i 的边表结点，表示从顶点 j 到顶点 i 有边。

(3) 对于邻接矩阵表示的无向图，顶点 i 的度等于第 i 行中元素为 1 的个数；对于邻接矩阵表示的有向图，顶点 i 的出度等于第 i 行中元素为 1 的个数，入度等于第 i 列中元素为 1 的个数，顶点 i 度等于它们之和。

对于邻接表 G 表示的无向图，顶点 i 的度等于 G->adjlist[i]为头结点的单链表中边表结点的个数。

对于邻接表 G 表示的有向图，顶点 i 的出度等于 G->adjlist[i]为头结点的单链表中边表结点的个数；入度需要遍历所有的边结点，若 G->adjlist[j]为头结点的单链表中存在编号为 i 的边结点，则顶点 i 的入度增 1，顶点 i 的度等于入度和出度之和。

5. 对于如图 8.3 所示的一个无向图 G，给出以顶点 0 作为初始点的所有的深度优先遍历序列和广度优先遍历序列。

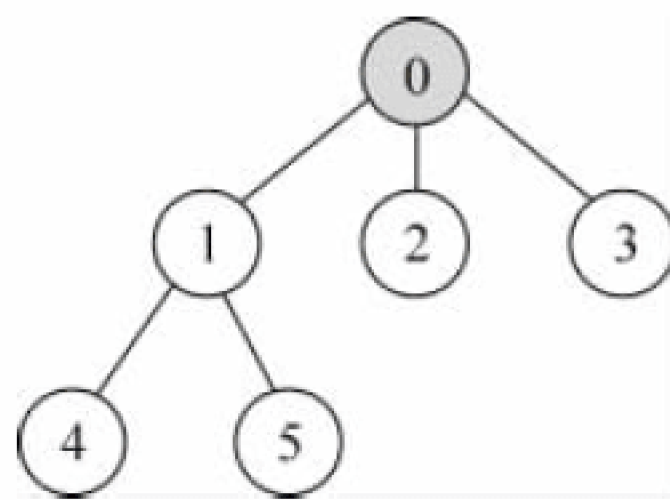


图 8.3 一个无向图 G

答：无向图 G 的所有深度优先遍历序列如下。

```
0 1 4 5 2 3
0 1 5 4 2 3
0 1 4 5 3 2
0 1 5 4 3 2
0 2 1 4 5 3
0 2 1 5 4 3
0 2 3 1 4 5
0 2 3 1 5 4
0 3 1 4 5 2
0 3 1 5 4 2
0 3 2 1 4 5
0 3 2 1 5 4
```

无向图 G 的所有广度优先遍历序列如下。

```
0 1 2 3 4 5
0 1 2 3 5 4
0 1 3 2 4 5
0 1 3 2 5 4
0 2 1 3 4 5
0 2 1 3 5 4
0 2 3 1 4 5
0 2 3 1 5 4
0 3 1 2 4 5
0 3 1 2 5 4
0 3 2 1 4 5
0 3 2 1 5 4
```

6. 对于如图 8.4 所示的带权无向图，给出利用 Prim 算法(从顶点 0 开始构造)和 Kruskal 算法构造出的最小生成树的结果，要求结果按构造边的顺序列出。

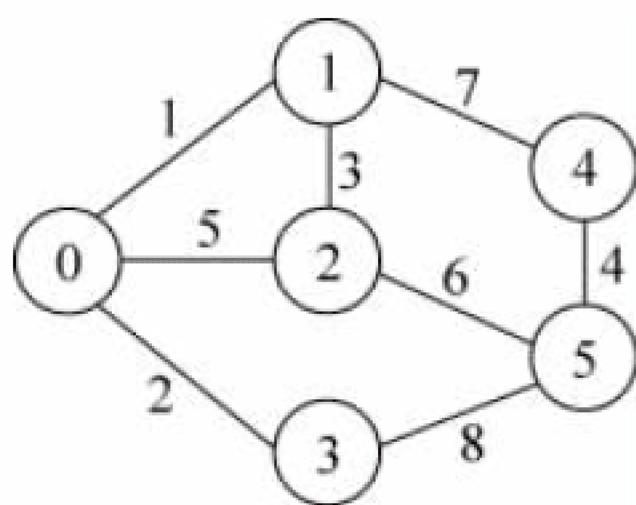


图 8.4　一个带权无向图

答：利用 Prim 算法从顶点 0 出发构造的最小生成树为{(0,1),(0,3),(1,2),(2,5),(5,4)},利用克鲁斯卡尔算法构造出的最小生成树为{(0,1),(0,3),(1,2),(5,4),(2,5)}。

7. 对于一个顶点个数超过 4 的带权无向图,回答以下问题:

(1) 该图的最小生成树一定是唯一的吗？如果所有边的权都不相同,那么其最小生成树一定是唯一的吗?

(2) 如果该图的最小生成树不是唯一的,那么调用 Prim 算法和 Kruskal 算法构造出的最小生成树一定相同吗?

(3) 如果图中有且仅有两条权最小的边,它们一定出现在该图的所有最小生成树中吗?简要说明理由。

(4) 如果图中有且仅有 3 条权最小的边,它们一定出现在该图的所有最小生成树中吗?简要说明理由。

答:(1) 该图的最小生成树不一定是唯一的。如果所有边的权都不相同,那么其最小生成树一定是唯一的。

(2) 若该图的最小生成树不是唯一的,那么调用 Prim 算法和 Kruskal 算法构造出的最小生成树不一定相同。

(3) 如果图中有且仅有两条权最小的边,它们一定会出现在该图的所有最小生成树中。因为在采用 Kruskal 算法构造最小生成树时首先选择这两条权最小的边加入,不会出现回路(严格的证明可以采用反证法)。

(4) 如果图中有且仅有 3 条权最小的边,它们不一定出现在该图的所有最小生成树中。因为在采用 Kruskal 算法构造最小生成树时,选择这 3 条权最小的边加入有可能出现回路。例如,如图 8.5 所示的带权无向图,有 3 条边的权均为 1,它们一定不会同时出现在其任何最小生成树中。

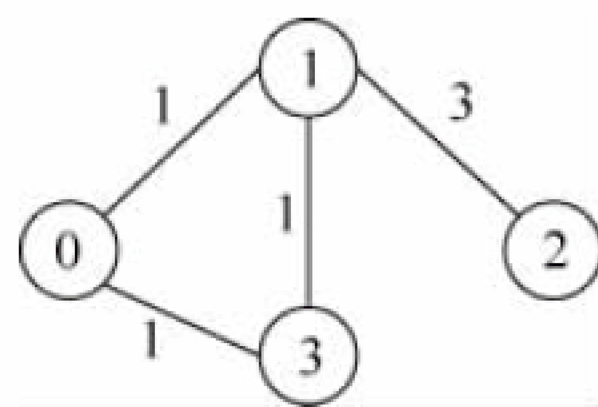


图 8.5　一个带权无向图

8. 对于如图 8.6 所示的带权有向图,采用 Dijkstra 算法求出从顶点 0 到其他各顶点的最短路径及其长度,要求给出求解过程。

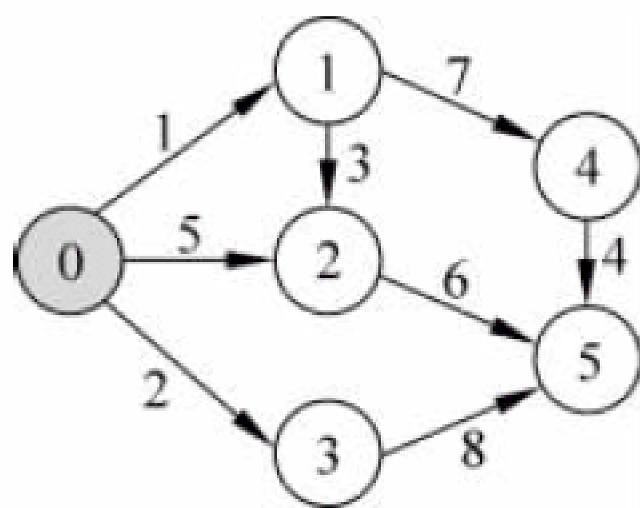


图 8.6　一个带权有向图 G

答:采用 Dijkstra 算法求从顶点 0 到其他各顶点的最短路径及其长度的过程如下。

(1) S={0},dist[0..5]={0,1,5,2,∞,∞},path[0..5]={0,0,0,0,−1,−1}。选取最短路径长度的顶点 1。

(2) S={0,1},调整顶点 1 到顶点 2、4 的最短路径长度,dist[0..5]={0,1,**4**,2,**8**,∞},path[0..5]={0,0,**1**,0,**1**,−1}。选取最短路径长度的顶点 3。

(3) S={0,1,3},调整顶点 3 到顶点 5 的最短路径长度,dist[0..5]={0,1,4,2,8,**10**},path[0..5]={0,0,1,0,1,**3**}。选取最短路径长度的顶点 2。

(4) S={0,1,3,2},调整顶点 2 到顶点 5 的最短路径长度,dist[0..5]={0,1,4,2,8,10},path[0..5]={0,0,1,0,1,3}。选取最短路径长度的顶点 4。

(5) S={0,1,3,2,4},调整顶点 4 到顶点 5 的最短路径长度,dist[0..5]={0,1,4,2,8,10},path[0..5]={0,0,1,0,1,3}。选取最短路径长度的顶点 5。

(6) $S=\{0,1,3,2,4,5\}$,顶点 5 没有出边,dist[0..5]={0,1,4,2,8,10},path[0..5]={0,0,1,0,1,3}。

最终结果如下:

```
从 0 到 1 的最短路径长度为:1,路径为:0,1
从 0 到 2 的最短路径长度为:4,路径为:0,1,2
从 0 到 3 的最短路径长度为:2,路径为:0,3
从 0 到 4 的最短路径长度为:8,路径为:0,1,4
从 0 到 5 的最短路径长度为:10,路径为:0,3,5
```

9. 对于一个带权连通图,可以采用 Prim 算法构造出从某个顶点 v 出发的最小生成树,问该最小生成树是否一定包含从顶点 v 到其他所有顶点的最短路径。如果回答是,请予以证明;如果回答不是,请给出反例。

答:不一定。例如,对于如图 8.7(a)所示的带权连通图,从顶点 0 出发的最小生成树如图 8.7(b)所示,而从顶点 0 到顶点 2 的最短路径为 0→2,不是最小生成树中的 0→1→2。

10. 若只求带权有向图 G 中从顶点 i 到顶点 j 的最短路径,如何修改 Dijkstra 算法来实现这一功能?

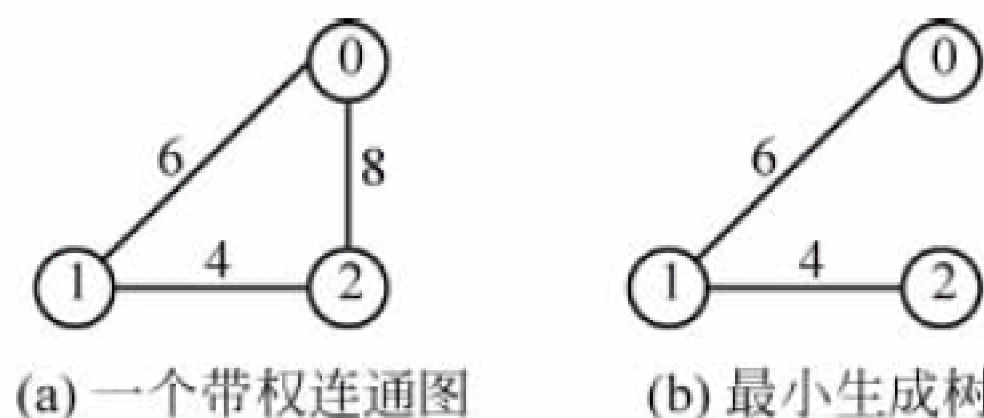


图 8.7 一个带权连通图及其最小生成树

答:修改 Dijkstra 算法为从顶点 i 开始(以顶点 i 为源点),按 Dijkstra 算法思路不断地扩展顶点集 S,当扩展到顶点 j 时算法结束,通过 path 回推出从顶点 i 到顶点 j 的最短路径。

11. Dijkstra 算法用于求单源最短路径,为了求一个图中所有顶点对之间的最短路径,可以以每个顶点作为源点调用 Dijkstra 算法,Floyd 算法和这种算法相比有什么优势?

答:对于有 n 个顶点的图,求所有顶点对之间的最短路径,若调用 Dijkstra 算法 n 次,其时间复杂度为 $O(n^3)$。Floyd 算法的时间复杂度也是 $O(n^3)$。但 Floyd 算法更快,这是因为前者每次调用 Dijkstra 算法时都是独立执行的,路径比较中得到的信息没有共享,而 Floyd 算法中每考虑一个顶点时所得到的路径比较信息保存在 A 数组中,会用于下次的路径比较,从而提高了整体查找最短路径的效率。

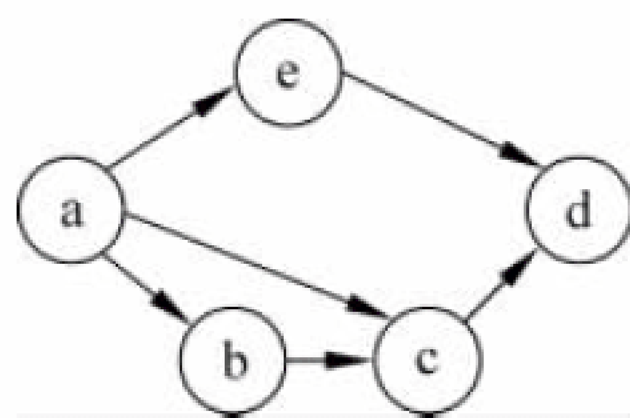


图 8.8 一个有向图

12. 回答以下有关拓扑排序的问题:

(1) 给出如图 8.8 所示有向图的所有不同的拓扑序列。

(2) 什么样的有向图的拓扑序列是唯一的?

(3) 现要对一个有向图的所有顶点重新编号,使所有表示边的非 0 元素集中到邻接矩阵数组的上三角部分。根据什么顺序对顶点进行编号可以实现这个功能?

答:(1) 该有向图的所有不同的拓扑序列有 aebcd、abced、abecd。

(2) 这样的有向图的拓扑序列是唯一的:图中只有一个入度为 0 的顶点,在拓扑排序中每次输出一个顶点后都只有一个入度为 0 的顶点。

(3) 首先对有向图进行拓扑排序,把所有顶点排在一个拓扑序列中;然后按该序列对所有顶点重新编号,使得每条有向边的起点编号小于终点编号,这样就可以把所有边集中到

邻接矩阵数组的上三角部分。

13. 已知有 6 个顶点(顶点编号为 0～5)的带权有向图 G,其邻接矩阵数组 **A** 为上三角矩阵,按行为主序(行优先)保存在以下的一维数组中：

4	6	∞	∞	∞	5	∞	∞	∞	4	3	∞	∞	3	3

要求：

(1) 写出图 G 的邻接矩阵数组 **A**。

(2) 画出带权有向图 G。

(3) 求图 G 的关键路径,并计算该关键路径的长度。

答：(1) 图 G 的邻接矩阵数组 **A** 如图 8.9 所示。

(2) 有向带权图 G 如图 8.10 所示。

(3) 图 8.11 中粗线所标识的 4 个活动组成图 G 的关键路径。

$$\boldsymbol{A}=\begin{bmatrix} 0 & 4 & 6 & \infty & \infty & \infty \\ \infty & 0 & 5 & \infty & \infty & \infty \\ \infty & \infty & 0 & 4 & 3 & \infty \\ \infty & \infty & \infty & 0 & \infty & 3 \\ \infty & \infty & \infty & \infty & 0 & 3 \\ \infty & \infty & \infty & \infty & \infty & 0 \end{bmatrix}$$

图 8.9 邻接矩阵 **A**

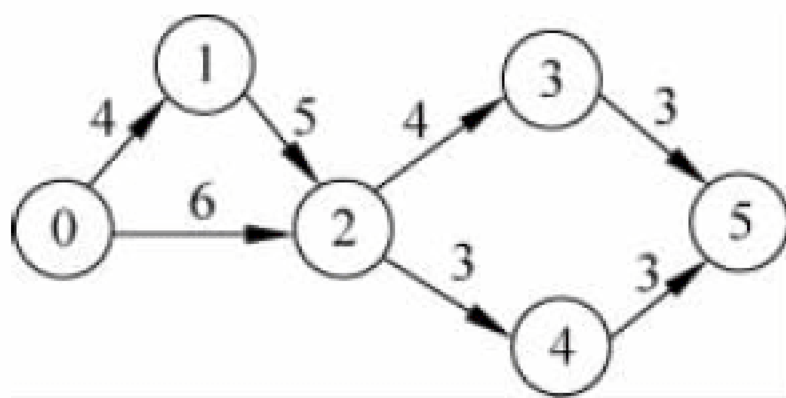


图 8.10 图 G

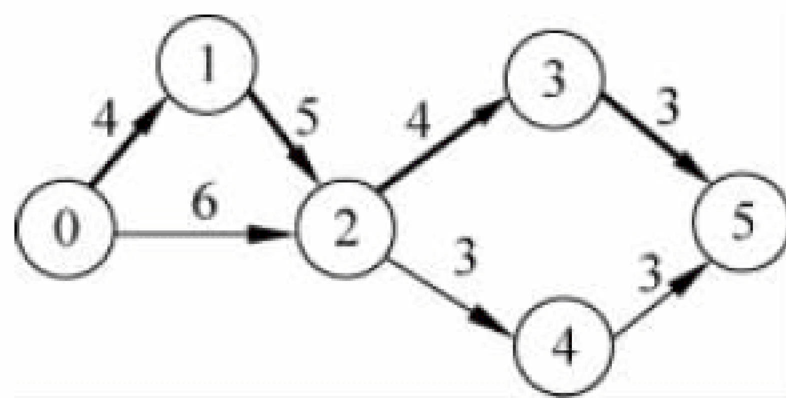


图 8.11 图 G 中的关键路径

14. 假设不带权有向图采用邻接矩阵 **g** 存储,设计实现以下功能的算法：

(1) 求出图中每个顶点的入度。

(2) 求出图中每个顶点的出度。

(3) 求出图中出度为 0 的顶点数。

解：利用邻接矩阵的特点和相关概念得到以下算法。

```
void InDs1(MatGraph g)                    //求出图 G 中每个顶点的入度
{   int i,j,n;
    printf("各顶点入度:\n");
    for (j=0;j<g.n;j++)
    {   n=0;
        for (i=0;i<g.n;i++)
            if (g.edges[i][j]!=0)
                n++;                      //n 累计入度数
        printf("  顶点%d:%d\n",j,n);
    }
}
void OutDs1(MatGraph g)                   //求出图 G 中每个顶点的出度
{   int i,j,n;
    printf("各顶点出度:\n");
    for (i=0;i<g.n;i++)
    {   n=0;
```

```
        for (j = 0;j < g.n;j++)
            if (g.edges[i][j]!= 0)
                n++;                         //n 累计出度数
        printf("  顶点 %d: %d\n",i,n);
    }
}
void ZeroOutDs1(MatGraph g)                  //求出图 G 中出度为 0 的顶点个数
{   int i,j,n;
    printf("出度为 0 的顶点:");
    for (i = 0;i < g.n;i++)
    {   n = 0;
        for (j = 0;j < g.n;j++)
            if (g.edges[i][j]!= 0)           //存在一条出边
                n++;
        if (n == 0)
            printf(" %2d\n",i);
    }
    printf("\n");
}
```

15. 假设不带权有向图采用邻接表 G 存储，设计实现以下功能的算法：

(1) 求出图中每个顶点的入度。

(2) 求出图中每个顶点的出度。

(3) 求出图中出度为 0 的顶点数。

解：利用邻接表的特点和相关概念得到以下算法。

```
void InDs2(AdjGraph * G)                //求出图 G 中每个顶点的入度
{   ArcNode * p;
    int A[MAXV],i;                      //A 存放各顶点的入度
    for (i = 0;i < G -> n;i++)          //A 中元素置初值 0
        A[i] = 0;
    for (i = 0;i < G -> n;i++)          //扫描所有头结点
    {   p = G -> adjlist[i].firstarc;
        while (p!= NULL)                //扫描边结点
        {   A[p -> adjvex]++;           //表示 i 到 p -> adjvex 顶点有一条边
            p = p -> nextarc;
        }
    }
    printf("各顶点入度:\n");             //输出各顶点的入度
    for (i = 0;i < G -> n;i++)
        printf("  顶点 %d: %d\n",i,A[i]);
}
void OutDs2(AdjGraph * G)               //求出图 G 中每个顶点的出度
{   int i,n;
    ArcNode * p;
    printf("各顶点出度:\n");
    for (i = 0;i < G -> n;i++)          //扫描所有头结点
    {   n = 0;
```

```
            p = G -> adjlist[i].firstarc;
            while (p!= NULL)          //扫描边结点
            {   n++;                  //累计出边的数
                p = p -> nextarc;
            }
            printf("   顶点 %d: %d\n", i, n);
        }
}
void ZeroOutDs2(AdjGraph *G)      //求出图 G 中出度为 0 的顶点数
{   int i, n;
    ArcNode *p;
    printf("出度为 0 的顶点:");
    for (i = 0; i < G -> n; i++)      //扫描所有头结点
    {   p = G -> adjlist[i].firstarc;
        n = 0;
        while (p!= NULL)          //扫描边结点
        {   n++;                  //累计出边的数
            p = p -> nextarc;
        }
        if (n == 0)               //输出边数为 0 的顶点编号
            printf(" %2d", i);
    }
    printf("\n");
}
```

16. 假设一个连通图采用邻接表作为存储结构，试设计一个算法，判断其中是否存在经过顶点 v 的回路。

解：从顶点 v 出发进行深度优先遍历，用 d 记录走过的路径长度，对每个访问的顶点设置标记为 1。若当前访问顶点 u，表示 $v \Rightarrow u$ 存在一条路径，如果顶点 u 的邻接点 w 等于 v 并且 $d > 1$，表示顶点 u 到 v 有一条边，即构成经过顶点 v 的回路，如图 8.12 所示。Cycle 算法中 has 是布尔值，初始调用时置为 false，执行后若为 true 表示存在经过顶点 v 的回路，否则表示没有相应的回路。

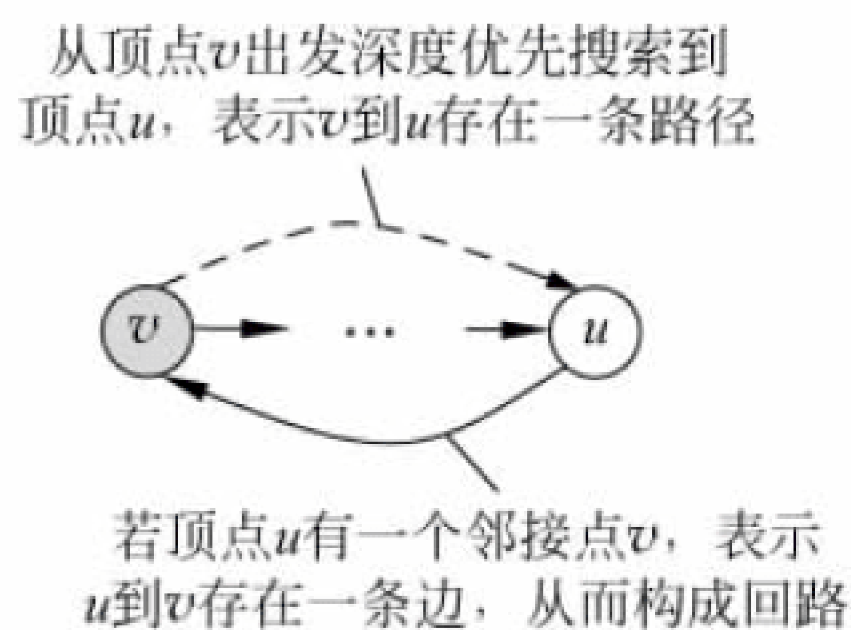


图 8.12　图中存在回路的示意图

对应的算法如下：

```
int visited[MAXV];                           //全局变量数组
void Cycle(AdjGraph *G, int u, int v, int d, bool &has)
{   //调用时 has 置初值 false, d 为 -1
    ArcNode *p; int w;
```

```
        visited[u] = 1; d++;                      //置已访问标记
        p = G -> adjlist[u].firstarc;             //p指向顶点u的第一个邻接点
        while (p!= NULL)
        {   w = p -> adjvex;
            if (visited[w] == 0)                  //若顶点w未访问,递归访问它
                Cycle(G,w,v,d,has);               //从顶点w出发搜索
            else if (w == v && d > 1)             //u到v存在一条边且回路长度大于1
            {   has = true;
                return;
            }
            p = p -> nextarc;                     //找下一个邻接点
        }
    }
    bool hasCycle(AdjGraph * G,int v)             //判断连通图G中是否有经过顶点v的回路
    {   bool has = false;
        Cycle(G,v,v,-1,has);                      //从顶点v出发搜索
        return has;
    }
```

17. 假设图 G 采用邻接表存储,试设计一个算法,判断无向图 G 是否为一棵树。若为树,返回真;否则返回假。

解:一个无向图 G 是一棵树的条件是 G 必须是无回路的连通图或者是有 $n-1$ 条边的连通图。这里采用后者作为判断条件,通过深度优先遍历图 G,并求出遍历过的顶点数 vn 和边数 en,若 vn==G->n 成立(表示为连通图)且 en==2(G->$n-1$)(遍历边数为 2(G->$n-1$))成立,则 G 为一棵树。对应的算法如下:

```
    void DFS2(AdjGraph * G,int v,int &vn,int &en)
    {   //深度优先遍历图G,并求出遍历过的顶点数vn和边数en
        ArcNode * p;
        visited[v] = 1;vn++;              //遍历过的顶点数增1
        p = G -> adjlist[v].firstarc;
        while (p!= NULL)
        {   en++;                         //遍历过的边数增1
            if (visited[p -> adjvex] == 0)
                DFS2(G,p -> adjvex,vn,en);
            p = p -> nextarc;
        }
    }
    int IsTree(AdjGraph * G)              //判断无向图G是否为一棵树
    {   int vn = 0,en = 0,i;
        for (i = 0;i < G -> n;i++)
            visited[i] = 0;
        DFS2(G,1,vn,en);
        if (vn == G -> n && en == 2 * (G -> n - 1))
            return 1;                     //遍历顶点为G->n个,遍历边数为2(G->n-1),则为树
        else
            return 0;
    }
```

18. 设5地(0~4)之间架设有6座桥(A~F),如图8.13所示,设计一个算法,从某地出发,经过每座桥恰巧一次,最后仍回到原地。

解：该实地图对应的一个无向图 G 如图 8.14 所示，本题变为从指定点 k 出发找经过所有 6 条边回到 k 顶点的路径，由于所有顶点的度均为偶数，可以找到这样的路径。

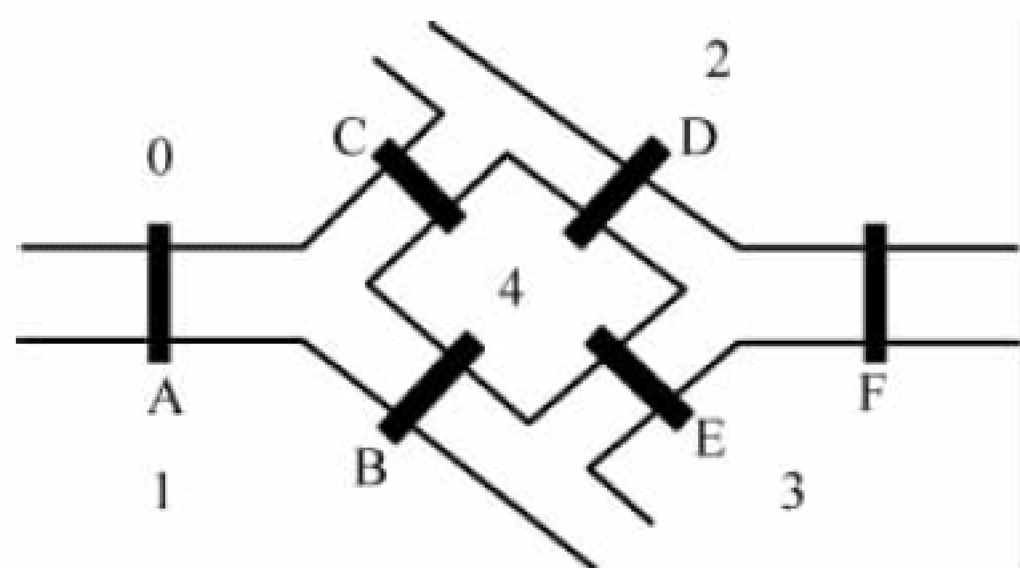


图 8.13　实地图

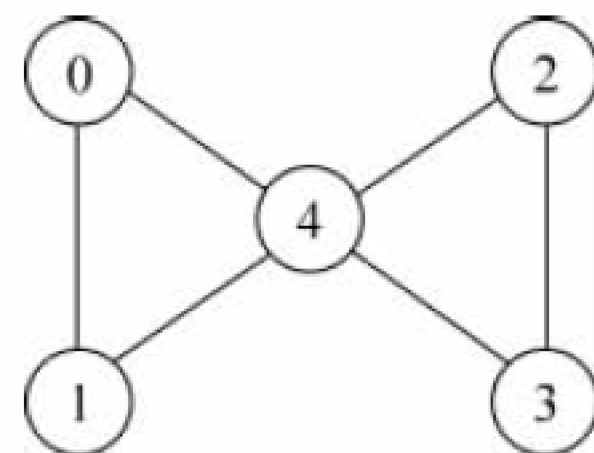


图 8.14　一个无向图 G

对应的算法如下：

```
int vedge[MAXV][MAXV];                //边访问数组,vedge[i][j]表示(i,j)边是否访问过
void Traversal(AdjGraph *G,int u,int v,int k,int path[],int d)
//d是到当前为止已走过的路径长度,调用时初值为-1
{   int w,i;
    ArcNode *p;
    d++; path[d]=v;                   //(u,v)加入到path中
    vedge[u][v]=vedge[v][u]=1;        //(u,v)边已访问
    p=G->adjlist[v].firstarc;         //p指向顶点v的第一条边
    while (p!=NULL)
    {   w=p->adjvex;                  //(v,w)有一条边
        if (w==k && d==G->e-1)        //找到一个回路,输出之
        {   printf("  %d->",k);
            for (i=0;i<=d;i++)
                printf("%d->",path[i]);
            printf("%d\n",w);
        }
        if (vedge[v][w]==0)           //(v,w)未访问过,则递归访问之
            Traversal(G,v,w,k,path,d);
        p=p->nextarc;                 //找v的下一条边
    }
    vedge[u][v]=vedge[v][u]=0;        //恢复环境:使该边点可重新使用
}
void FindCPath(AdjGraph *G,int k)     //输出经过顶点k和所有边的全部回路
{   int path[MAXV];
    int i,j,v;
    ArcNode *p;
    for (i=0;i<G->n;i++)              //vedge数组置初值
        for (j=0;j<G->n;j++)
            if (i==j) vedge[i][j]=1;
            else vedge[i][j]=0;
    printf("经过顶点%d的走过所有边的回路:\n",k);
    p=G->adjlist[k].firstarc;
    while (p!=NULL)
    {   v=p->adjvex;
        Traversal(G,k,v,k,path,-1);
        p=p->nextarc;
    }
}
```

设计以下主函数：

```
int main()
{	int v = 4;
	AdjGraph * G;
	int n = 5,e = 6;
	int A[MAXV][MAXV] = {{0,1,0,0,1},{1,0,0,0,1},
					{0,0,0,1,1},{0,0,1,0,1},{1,1,1,1,0}};
	CreateAdj(G,A,n,e);
	printf("图 G 的邻接表:\n");DispAdj(G);		//输出邻接表
	FindCPath(G,v);
	printf("\n");
	DestroyAdj(G);
	return 1;
}
```

程序的执行结果如下：

```
图 G 的邻接表:
  0:   1[1]→   4[1]→∧
  1:   0[1]→   4[1]→∧
  2:   3[1]→   4[1]→∧
  3:   2[1]→   4[1]→∧
  4:   0[1]→   1[1]→   2[1]→   3[1]→∧
经过顶点 4 的走过所有边的回路:
  4->0->1->4->2->3->4
  4->0->1->4->3->2->4
  4->1->0->4->2->3->4
  4->1->0->4->3->2->4
  4->2->3->4->0->1->4
  4->2->3->4->1->0->4
  4->3->2->4->0->1->4
  4->3->2->4->1->0->4
```

19. 设不带权无向图 G 采用邻接表表示，设计一个算法求源点 i 到其余各顶点的最短路径。

解：利用广度优先遍历的思想，求 i 和 j 两顶点间的最短路径转化为求从 i 到 j 的层数，为此设计一个 level[]数组记录每个顶点的层次。对应的算法如下：

```
void ShortPath(AdjGraph * G,int i)
{	int qu[MAXV],level[MAXV];
	int front = 0,rear = 0,k,lev;			//lev 保存从 i 到访问顶点的层数
	ArcNode * p;
	visited[i] = 1;
	rear++;qu[rear] = i;level[rear] = 0;		//顶点 i 已访问,将其进队
	while (front!= rear)				//队非空则执行
	{	front = (front + 1) % MAXV;
		k = qu[front];				//出队
		lev = level[front];
		if (k!= i)
```

```
            printf("顶点%d到顶点%d的最短距离是:%d\n",i,k,lev);
        p=G->adjlist[k].firstarc;           //取k的边表头指针
        while (p!=NULL)                     //依次搜索邻接点
        {   if (visited[p->adjvex]==0)      //若未访问过
            {   visited[p->adjvex]=1;
                rear=(rear+1)%MAXV;
                qu[rear]=p->adjvex;         //访问过的邻接点进队
                level[rear]=lev+1;
            }
            p=p->nextarc;                   //找顶点i的下一邻接点
        }
    }
}
```

设计以下主函数：

```
int main()
{   AdjGraph *G;
    int n=5,e=8;
    int A[MAXV][MAXV]={{0,1,0,1,1},{1,0,1,1,0},
        {0,1,0,1,1},{1,1,1,0,1},{1,0,1,1,0}};
    CreateAdj(G,A,n,e);                      //创建《教程》中图8.1(a)所示的邻接表
    printf("图G的邻接表:\n");DispAdj(G);     //输出邻接表
    for (int i=0;i<n;i++)
        visited[i]=0;
    printf("顶点1到其他各顶点的最短距离如下:\n");
    ShortPath(G,1);
    return 1;
}
```

程序的执行结果如下：

```
图G的邻接表:
  0:   1[1]→  3[1]→  4[1]→∧
  1:   0[1]→  2[1]→  3[1]→∧
  2:   1[1]→  3[1]→  4[1]→∧
  3:   0[1]→  1[1]→  2[1]→  4[1]→∧
  4:   0[1]→  2[1]→  3[1]→∧
顶点1到其他各顶点的最短距离如下:
  顶点1到顶点0的最短距离是:1
  顶点1到顶点2的最短距离是:1
  顶点1到顶点3的最短距离是:1
  顶点1到顶点4的最短距离是:2
```

20. 对于一个带权有向图，设计一个算法输出从顶点 i 到顶点 j 的所有路径及其路径长度。调用该算法求出《教程》的图8.35中顶点0到顶点3的所有路径及其长度。

解：采用回溯的深度优先遍历方法。增加一个形参length表示路径长度，其初始值为

0。当从顶点 u 出发，设置 visited[u]=1，当找到一个没有访问过的邻接点 w，就从 w 出发递归查找，其路径长度 length 增加 $<u,w>$ 边的权值。当找到终点 v，就输出一条路径。通过设置 visited[u]=0 回溯查找所有的路径。对应的算法如下：

```
int visited[MAXV];
void findpath(AdjGraph *G,int u,int v,int path[],int d,int length)
{   //d表示path中顶点个数,初始为0; length表示路径长度,初始为0
    int w,i;
    ArcNode *p;
    path[d]=u; d++;                    //顶点u加入到路径中,d增1
    visited[u]=1;                      //置已访问标记
    if (u==v && d>0)                   //找到一条路径则输出
    {   printf("  路径长度:%d, 路径:",length);
        for (i=0;i<d;i++)
            printf("%2d",path[i]);
        printf("\n");
    }
    p=G->adjlist[u].firstarc;          //p指向顶点u的第一个邻接点
    while (p!=NULL)
    {   w=p->adjvex;                   //w为顶点u的邻接点
        if (visited[w]==0)             //若w顶点未访问,递归访问它
            findpath(G,w,v,path,d,p->weight+length);
        p=p->nextarc;                  //p指向顶点u的下一个邻接点
    }
    visited[u]=0;                      //恢复环境,使该顶点可重新使用
}
```

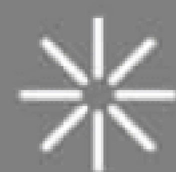
设计以下主函数求《教程》的图 8.35 中顶点 0 到顶点 3 的所有路径及其长度。

```
int main()
{   AdjGraph *G;
    int A[MAXV][MAXV]={
        {0,4,6,6,INF,INF,INF},{INF,0,1,INF,7,INF,INF},
        {INF,INF,0,INF,6,4,INF},{INF,INF,2,0,INF,5,INF},
        {INF,INF,INF,INF,0,INF,6},{INF,INF,INF,INF,1,0,8},
        {INF,INF,INF,INF,INF,INF,0}};
    int n=7, e=12;
    CreateAdj(G,A,n,e);                          //创建《教程》中图8.35的邻接表
    printf("图G的邻接表:\n");
    DispAdj(G);                                  //输出邻接表
    int u=0,v=5;
    int path[MAXV];
    printf("从%d->%d的所有路径:\n",u,v);
    findpath(G,u,v,path,0,0);
    DestroyAdj(G);
    return 1;
}
```

上述程序的执行结果如下：

```
图 G 的邻接表:
  0:  1[4]→ 2[6] → 3[6] → ∧
  1:  2[1] → 4[7] → ∧
  2:  4[6] → 5[4] → ∧
  3:  2[2] → 5[5] → ∧
  4:  6[6] → ∧
  5:  4[1] → 6[8] → ∧
  6: ∧
从 0->5 的所有路径:
  路径长度:9, 路径: 0 1 2 5
  路径长度:10, 路径:0 2 5
  路径长度:12, 路径:0 3 2 5
  路径长度:11, 路径:0 3 5
```

8.3 补充练习题及参考答案



8.3.1 单项选择题

1. 所谓简单路径是指除了起点和终点以外________。
   A. 任何一条边在这条路径上不重复出现
   B. 任何一个顶点在这条路径上不重复出现
   C. 这条路径由一个顶点序列构成，不包含边
   D. 这条路径由边序列构成，不包含顶点

答：简单路径由顶点序列组成，且其中除了起点和终点以外其他顶点不重复出现。本题的答案为 B。

2. 带权有向图 G 用邻接矩阵 $\boldsymbol{A}$ 存储，则顶点 i 的入度等于 A 中________。
   A. 第 i 行非∞的元素之和　　B. 第 i 列非∞的元素之和
   C. 第 i 行非∞且非 0 的元素个数　　D. 第 i 列非∞且非 0 的元素个数

答：在带权有向图的邻接矩阵中，元素 0 和∞表示的都不是有向边，而入度是由邻接矩阵的列中的元素个数计算出来的。本题的答案为 D。

3. 无向图的邻接矩阵是一个________。
   A. 对称矩阵　　B. 零矩阵　　C. 上三角矩阵　　D. 对角矩阵

答：A。

4. 在一个无向图中，所有顶点的度之和等于边数的________倍。
   A. 1/2　　B. 1　　C. 2　　D. 4

答：在无向图中，一条边计入两个顶点的度数。本题的答案为 C。

5. 一个有 n 个顶点的无向图最多有________条边。
   A. n　　B. $n(n-1)$　　C. $n(n-1)/2$　　D. $2n$

答：为完全无向图时边最多。本题的答案为 C。

6. 具有 6 个顶点的无向图至少应有________条边才可能是一个连通图。

A. 5　　B. 6　　C. 7　　D. 8

答：具有 n 个顶点的无向图，当边数少于 $n-1$ 时不可能是连通图，只有边数大于等于 $n-1$ 时才有可能是连通图。本题的答案为 A。

7. 若无向图 $G(V,E)$ 中含 7 个顶点，则保证图 G 在任何情况下都是连通的需要的边数最少是________。

A. 6　　B. 15　　C. 16　　D. 21

答：对于具有 n 个顶点的无向图，当其中 $n-1$ 个顶点构成一个完全图时，再加上任一条边必然构成一个连通图，所以最少边数 $=(n-1)(n-2)/2+1=16$。也就是说，对于含 7 个顶点的无向图，如果边数大于等于 16，无论怎么摆放这些边，它总是连通的。本题的答案为 C。

8. 设 G 是一个非连通无向图，有 15 条边，则该图至少有________个顶点。

A. 5　　B. 6　　C. 7　　D. 15

答：设图中有 n 个顶点，则 $e \leqslant n(n-1)/2$。这里 $e=15$，当 $e=n(n-1)/2$ 时，得到 $n=6$，也就是说当 $n=6$ 时该图是完全无向图，一定是连通的。要使该图不连通，n 应大于 6，最小值为 7。本题的答案为 C。

9. 设图 G 是一个含有 $n(n>1)$ 个顶点的连通图，其中任意一条简单路径的长度不会超过________。

A. 1　　B. n　　C. $n-1$　　D. $n/2$

答：最长的简单路径包含图中的所有顶点，其长度为 $n-1$。本题的答案为 C。

10. 下列关于无向连通图特征的叙述正确的是________。

Ⅰ. 所有顶点的度之和为偶数

Ⅱ. 边数大于顶点个数减 1

Ⅲ. 至少有一个顶点的度为 1

A. 只有Ⅰ　　B. 只有Ⅱ　　C. Ⅰ和Ⅱ　　D. Ⅰ和Ⅲ

答：在无向图中，一条边在度之和中计为 2，所以度之和为边数的两倍，Ⅰ正确。无向连通图边数最少时为树图的情况，此时边数为顶点个数减 1，所以Ⅱ错误。一个顶点数为 3，边数为 3 的无向连通图中所有顶点度为 2，所以Ⅲ错误。本题的答案为 A。

11. 在图 8.15 所示的有向图中存在一个强连通分量 $G=(V,E)$，其中，________。

A. $V=\{2,3,5,6\}$，$E=\{<5,2>,<2,3>,<2,6>,<6,3>,<3,5>\}$

B. $V=\{2,3,5,6\}$，$E=\{<5,2>,<2,6>,<6,3>,<3,5>\}$

C. $V=\{2,3,5\}$，$E=\{<2,3>,<3,5>,<5,2>\}$

D. $V=\{1,7\}$，$E=\{<1,7>,<7,1>\}$

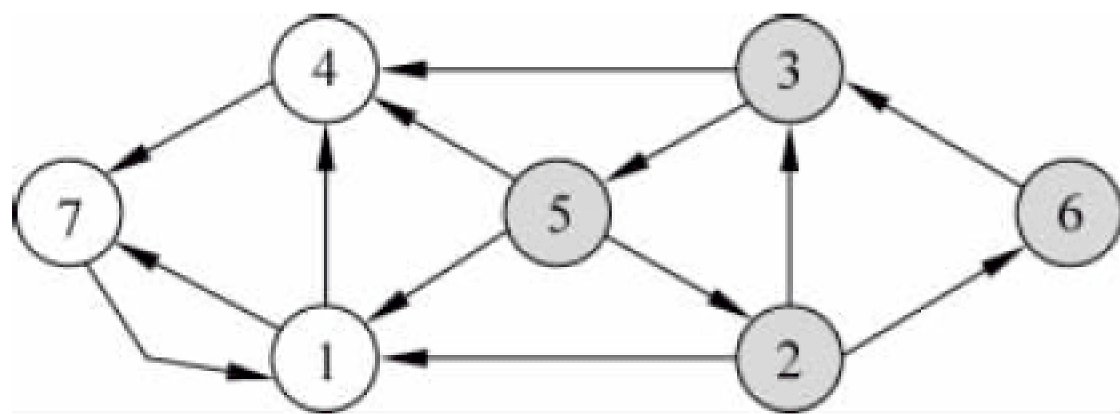


图 8.15　一个有向图

答：顶点 2→6→3→5→2 构成一个回路，它恰好是一个强连通分量。本题的答案为 A。

12. 下列________的邻接矩阵是对称矩阵。

A. 有向图　　B. 无向图　　C. AOV 网　　D. AOE 网

答：B。

13. 若图的邻接矩阵中主对角线上的元素全是 0，其余元素全是 1，则可以断定该图一定是________。

A. 无向图　　B. 非带权图　　C. 有向图　　D. 完全图

答：D。

14. 一个有向图 G 的邻接表存储如图 8.16 所示，现按深度优先搜索遍历，从顶点 0 出发，所得到的顶点序列是________。

A. 0,1,2,3,4　　B. 0,1,2,4,3　　C. 0,1,3,4,2　　D. 0,1,4,2,3

答：B。

15. 对于图 8.17 所示的无向图，从顶点 1 开始进行深度优先遍历，可得到顶点访问序列是________。

A. 1 2 4 3 5 7 6　　B. 1 2 4 3 5 6 7

C. 1 2 4 5 6 3 7　　D. 1 2 3 4 5 7 6

答：选项 B 中从顶点 5 到顶点 6 不符合优先遍历规则；选项 C 中从顶点 5 到顶点 6 不符合优先遍历规则；选项 D 中从顶点 2 到顶点 3 不符合优先遍历规则。本题的答案为 A。

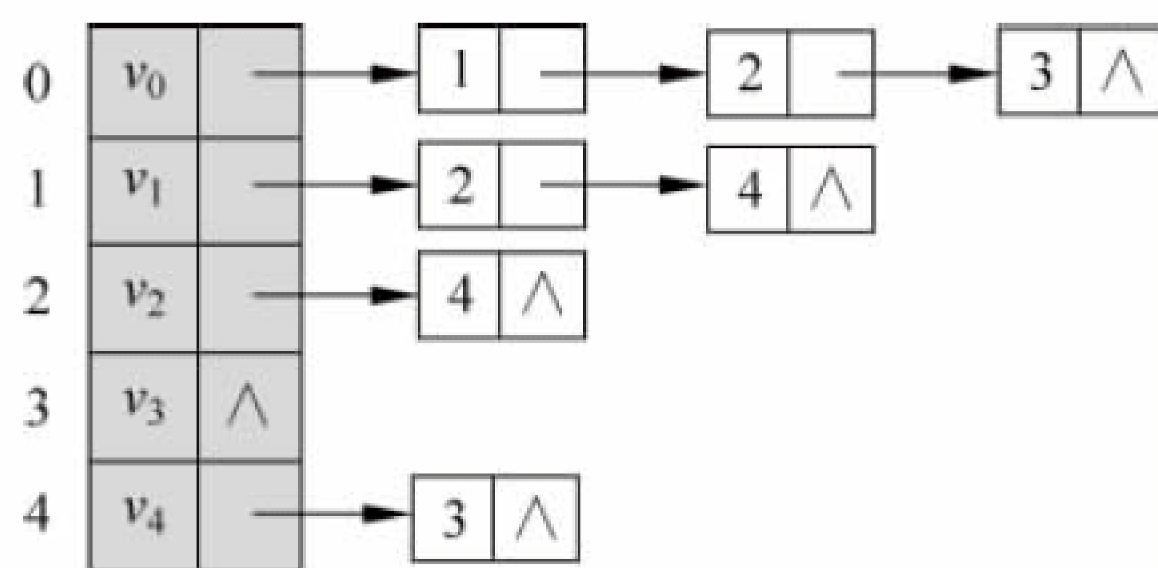


图 8.16　有向图 G 的邻接表

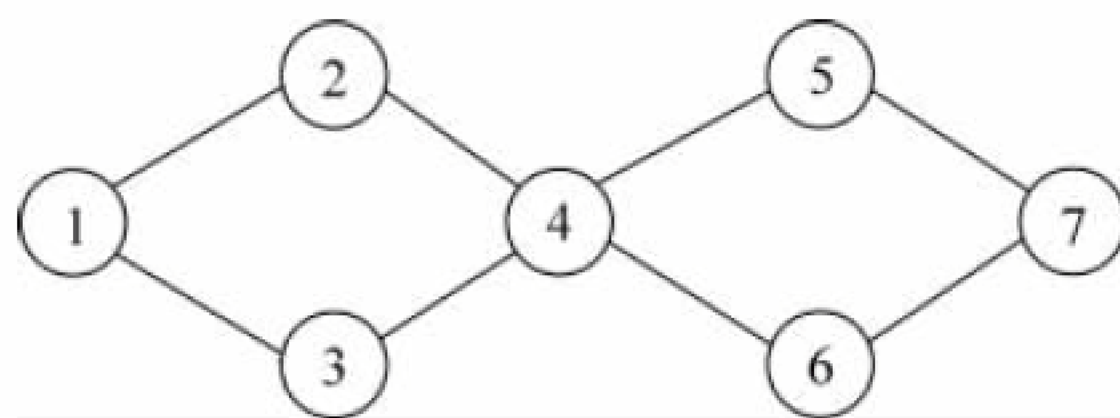


图 8.17　一个无向图

16. 对于图 8.17 所示的无向图，从顶点 1 开始进行广度优先遍历，可得到顶点访问序列是________。

A. 1 3 2 4 5 6 7　　B. 1 2 4 3 5 6 7

C. 1 2 3 4 5 7 6　　D. 2 5 1 4 7 3 6

答：选项 B 中 124 子序列不符合广度优先遍历规则；选项 C 中 457 子序列不符合广度优先遍历规则；选项 D 中从 2 开始不符合广度优先遍历规则。本题的答案为 A。

17. 如果从无向图的任一顶点出发进行一次深度优先遍历即可访问所有顶点，则该图一定是________。

A. 完全图　　B. 连通图　　C. 有回路　　D. 一棵树

答：B。

18. 采用邻接表存储的图的深度优先遍历算法类似于二叉树的________算法。

A. 先序遍历　　B. 中序遍历　　C. 后序遍历　　D. 层次遍历

答：A。

19. 采用邻接表存储的图的广度优先遍历算法类似于二叉树的________算法。
A. 先序遍历 B. 中序遍历 C. 后序遍历 D. 层次遍历

答：D。

20. 在图的广度优先遍历算法中用到一个队列，每个顶点最多进队________次。
A. 1 B. 2 C. 3 D. 不确定

答：A。

21. 以下关于广度优先遍历的叙述正确的是________。
A. 广度优先遍历不适合有向图
B. 对任何有向图调用一次广度优先遍历算法便可访问所有的顶点
C. 对一个强连通图调用一次广度优先遍历算法便可访问所有的顶点
D. 对任何非强连通图需要多次调用广度优先遍历算法才可访问所有的顶点

答：C。对于边集为{<0,1>,<1,2>}的非强连通图，从顶点 0 出发调用一次 BFS 算法也可以访问所有顶点，所以选项 D 是错误的。

22. 任何一个含两个或以上顶点的带权无向连通图________最小生成树。
A. 只有一棵 B. 有一棵或多棵
C. 一定有多棵 D. 可能不存在

答：B。

23. 一个无向连通图的生成树是含有该连通图的全部顶点的________。
A. 极小连通子图 B. 极小子图
C. 极大连通子图 D. 极大子图

答：A。

24. 设有无向图 $G=(V,E)$ 和 $G'=(V',E')$，如 G' 是 G 的生成树，则下面说法错误的是________。
A. G' 为 G 的连通分量 B. G' 是 G 的无环子图
C. G' 为 G 的子图 D. G' 为 G 的极小连通子图且 $V'=V$

答：选项 B、D 均为生成树的特点，而选项 A 为概念错误，G' 为连通子图而非连通分量，因为连通分量不一定是生成树。本题的答案为 A。

25. 对于有 n 个顶点的带权连通图，它的最小生成树是指图中任意一个________。
A. 由 $n-1$ 条权值最小的边构成的子图
B. 由 $n-1$ 条权值之和最小的边构成的子图
C. 由 n 个顶点构成的极大连通子图
D. 由 n 个顶点构成的极小连通子图，且边的权值之和最小

答：D。

26. 对某个带权连通图构造最小生成树，以下说法中正确的是________。
Ⅰ. 该图的所有最小生成树的总代价一定是唯一的
Ⅱ. 其所有权值最小的边一定会出现在所有的最小生成树中
Ⅲ. 用 Prim 算法从不同顶点开始构造的所有最小生成树一定相同
Ⅳ. 使用 Prim 算法和 Kruskal 算法得到的最小生成树总不相同
A. 仅Ⅰ B. 仅Ⅱ C. 仅Ⅰ、Ⅲ D. 仅Ⅱ、Ⅳ

答：A。

27. 用 Prim 算法求一个连通的带权图的最小代价生成树，在算法执行的某时刻，已选取的顶点集合 $U=\{1,2,3\}$，已选取的边的集合 $TE=\{(1,2),(2,3)\}$，要选取下一条权值最小的边，应当从________组中选取。

A. {(1,4),(3,4),(3,5),(2,5)}　　B. {(4,5),(1,3),(3,5)}

C. {(1,2),(2,3),(3,5)}　　D. {(3,4),(3,5),(4,5),(1,4)}

答：$U=\{1,2,3\}$，$V-U=\{4,5,\cdots\}$，候选边只能是这两个顶点集之间的边。本题的答案为 A。

28. 用 Prim 算法求一个连通的带权图的最小代价生成树，在算法执行的某时刻，已选取的顶点集合 $U=\{1,2,3\}$，边的集合 $TE=\{(1,2),(2,3)\}$，要选取下一条权值最小的边，不可能从________组中选取。

A. {(1,4),(3,4),(3,5),(2,5)}　　B. {(1,5),(2,4),(3,5)}

C. {(1,2),(2,3),(3,5)}　　D. {(1,4),(3,5),(2,5),(3,4)}

答：$U=\{1,2,3\}$，$V-U=\{4,5,\cdots\}$，候选边只能是这两个顶点集之间的边。本题的答案为 C。

29. 用 Kruskal 算法求一个连通的带权图的最小代价生成树，在算法执行的某时刻，已选取的边集合 $TE=\{(1,2),(2,3),(3,5)\}$，要选取下一条权值最小的边，可能选取的边是________。

A. (1,2)　　B. (3,5)　　C. (2,5)　　D. (6,7)

答：选取(1,2)、(3,5)、(2,5)边会构成回路。本题的答案为 D。

30. 在用 Prim 和 Kruskal 算法构造最小生成树时，前者更适合于___①___，后者更适合于___②___。

A. 有向图　　B. 无向图　　C. 稀疏图　　D. 稠密图

答：①D ②C。

31. 对含有 n 个顶点、e 条边的带权图求最短路径的 Dijkstra 算法的时间复杂度为________。

A. $O(n)$　　B. $O(n+e)$　　C. $O(n^2)$　　D. $O(ne)$

答：C。

32. Dijkstra 算法是________方法求出图中从某顶点到其余顶点最短路径的。

A. 按长度递减的顺序求出图的某顶点到其余顶点的最短路径

B. 按长度递增的顺序求出图的某顶点到其余顶点的最短路径

C. 通过深度优先遍历求出图中某顶点到其余顶点的最短路径

D. 通过广度优先遍历求出图中某顶点到其余顶点的最短路径

答：B。

33. 用 Dijkstra 算法求一个带权有向图 G 中从顶点 0 出发的最短路径，在算法执行的某时刻，$S=\{0,2,3,4\}$，下一步选取的目标顶点可能是________。

A. 顶点 2　　B. 顶点 3　　C. 顶点 4　　D. 顶点 7

答：下一步只能选取 V-S 中的顶点。本题的答案为 D。

34. 用 Dijkstra 算法求一个带权有向图 G 中从顶点 0 出发的最短路径，在算法执行的某时刻，$S=\{0,2,3,4\}$，选取的目标顶点是顶点 1，则可能修改的最短路径是________。

A. 从顶点 0 到顶点 2 的最短路径　　B. 从顶点 2 到顶点 4 的最短路径

C. 从顶点 0 到顶点 1 的最短路径　　D. 从顶点 0 到顶点 3 的最短路径

答：只可能修改从顶点 0 到 V-S 中的某个顶点的最短路径。本题的答案为 C。

35. 有一个顶点编号为 0～4 的带权有向图 G，现用 Floyd 算法求任意两个顶点之间的最短路径，在算法执行的某时刻已考虑了 0～2 的顶点，现考虑顶点 3，则以下叙述中正确的是________。

A. 只可能修改从顶点 0～2 到顶点 3 的最短路径

B. 只可能修改从顶点 3 到顶点 0～2 的最短路径

C. 只可能修改从顶点 0～2 到顶点 4 的最短路径

D. 所有两个顶点之间的路径都可能被修改

答：D。

36. 对如图 8.18 所示的有向带权图，若采用 Dijkstra 算法求源点 a 到其他各顶点的最短路径，则得到的第一条最短路径的目标顶点是 b，第二条最短路径的目标顶点是 c，后续得到的其余各最短路径的目标顶点依次是________。

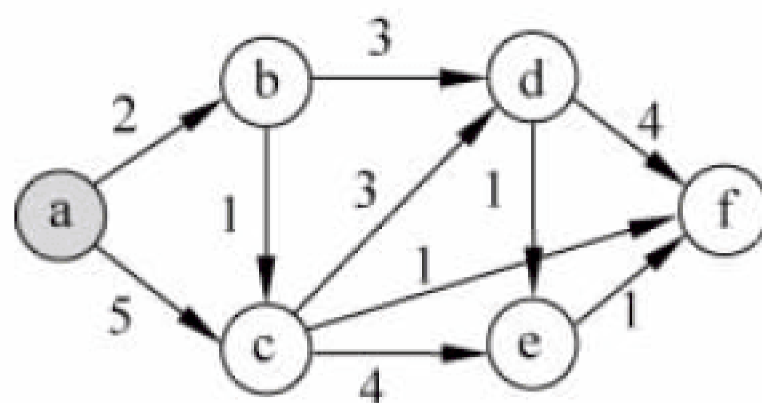


图 8.18　一个有向图

A. d，e，f　　B. e，d，f

C. f，d，e　　D. f，e，d

答：C。

37. 判定一个有向图是否存在回路除了可以利用拓扑排序方法以外，还可以用________。

A. 求关键路径的方法　　B. 求最短路径的 Dijkstra 方法

C. 广度优先遍历算法　　D. 深度优先遍历算法

答：利用拓扑排序算法可以判定有向图中是否存在回路。如果拓扑排序的结果只得到了部分顶点的拓扑有序序列，则该有向图中存在有回路。另外也可以利用深度优先遍历算法判定有向图 G 中是否存在回路，若深度优先遍历过程中遇到了回边则必定存在环。本题的答案为 D。

38. 若一个有向图中的顶点不能排成一个拓扑序列，则可断定该有向图________。

A. 是个有根有向图　　B. 是个强连通图

C. 含有多个入度为 0 的顶点　　D. 含有顶点数目大于 1 的强连通分量

答：该图中存在回路，该回路构成一个顶点个数大于 1 的强连通分量。本题的答案为 D。

39. 关键路径是 AOE 网中________。

A. 从源点到汇点的最长路径　　B. 从源点到汇点的最短路径

C. 最长的回路　　D. 最短的回路

答：在 AOE 网中，从源点到汇点的最长路径称为关键路径。本题的答案为 A。

40. 对于 AOE 网的关键路径，以下叙述中正确的是________。

A. 任何一个关键活动提前完成，则整个工程也会提前完成

B. 完成工程的最短时间是从源点到汇点的最短路径长度

C. 一个 AOE 网的关键路径是唯一的

D. 任何一个活动持续时间的改变可能会影响关键路径的改变

答：D。

8.3.2 填空题

1. 有 n 个顶点的无向图最多有________条边。

答：为完全无向图时边数最多，本题的答案为 $n(n-1)/2$。

2. 有 n 个顶点的强连通有向图 G 至少有________条边。

答：n。n 个顶点的有向图依次首尾相连构成一个环，此时为边数最少的强连通图。

3. 在有 n 个顶点的有向图中，每个顶点的度最大可达________。

答：$2(n-1)$。

4. 一个图的___①___存储结构是唯一的，而___②___存储结构不一定是唯一的。

答：图的存储结构主要有邻接矩阵和邻接表，前者唯一，后者不一定唯一。本题的答案为①邻接矩阵　②邻接表。

5. 用邻接矩阵 **A** 存储不带权有向图 G，其第 i 行的所有元素之和等于顶点 i 的________。

答：出度。

6. 有 n 个顶点的有向图 G 最多有________条边。

答：$n(n-1)$。为完全有向图时边数最多。

7. 对于一个具有 n 个顶点、e 条边的无向图，若采用邻接表表示，则头结点数组的大小为___①___，边结点总数是___②___。

答：①n　②$2e$。

8. 已知一个有向图采用邻接矩阵表示，删除所有从第 i 个顶点出发的边的操作是________。

答：将邻接矩阵的第 i 行全部置为 0。

9. 对于 n 个顶点的不带权无向图，采用邻接矩阵表示，求图中边数的方法是___①___，判断任意两个顶点 i 和 j 是否有边相连的方法是___②___，求任意一个顶点的度的方法是___③___。

答：①邻接矩阵中 1 的个数除以 2　②A[i][j]是否为 1　③计算该行或该列中 1 的个数。

10. 对于 n 个顶点的不带权有向图，采用邻接矩阵表示，求图中边数的方法是___①___，判断顶点 i 到顶点 j 是否有边的方法是___②___，求任意一个顶点的度的方法是___③___。

答：① 邻接矩阵中 1 的个数　② A[i][j]是否为 1　③ 出度为该行中 1 的个数，入度为该列中 1 的个数，顶点的度等于出度和入度之和。

11. 对于 n 个顶点的无向图，采用邻接表表示，求图中边数的方法是___①___，判断任意两个顶点 i 和 j 是否有边相连的方法是___②___，求任意一个顶点的度的方法是___③___。

答：①邻接表中边结点个数除以 2　②顶点 i 对应的第 i 个单链表中是否包含 j 结点　③顶点 i 对应的第 i 个单链表中的结点个数。

12. 无向图的连通分量是指________。

答：极大连通子图。

13. 一个有 n 个顶点、e 条边的连通图采用邻接表表示，从某个顶点 v 出发进行深度优先遍历 DFS(G,v)，则最大的递归深度是________。

答：n。例如，$n=4$，边集为{(0,1),(1,2),(2,3)}，$v=0$，则 DFS(G,0)→DFS(G,1)→DFS(G,2)→DFS(G,3)，递归深度为 4。

14. 一个有 n 个顶点、e 条边的连通图采用邻接表表示，从某个顶点 v 出发进行广度优先遍历 BFS(G,v)，则队列中最多的顶点个数是________。

答：$n-1$。如果图中顶点 v 有 $n-1$ 个相邻点，这 $n-1$ 个相邻点都会进队。

15. 有 n 个顶点、e 条边的图 G 采用邻接矩阵表示，从顶点 v 出发进行深度优先遍历的时间复杂度为________。

答：$O(n^2)$。

16. 对于 n 个顶点的连通图来说，它的生成树一定有________条边。

答：$n-1$。

17. 若含有 n 个顶点的无向图恰好形成一个环，则它有________棵生成树。

答：n。该图有 n 条边，删除任一条边得到一棵生成树。

18. 一个连通图的________是一个极小连通子图。

答：生成树。

19. Prim 算法适用于求____①____的网的最小生成树，Kruskal 算法适用于求____②____的网的最小生成树。

答：① 边稠密　② 边稀疏。

20. 对于如图 8.19 所示的带权有向图，采用 Dijkstra 算法求从顶点 0 到其他顶点的最短路径，当考虑的当前顶点为顶点 3 时，可能修改的最短路径的顶点是________。

答：4 和 5。因为图中存在<3,4>和<3,5>的边。

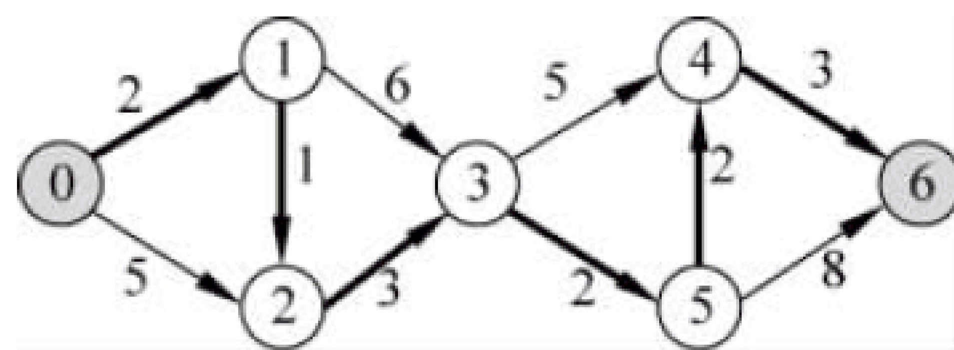


图 8.19　一个带权有向图

21. 对于如图 8.19 所示的带权有向图，采用 Dijkstra 算法求从顶点 0 到顶点 6 的最短路径，该路径是________。

答：0→1→2→3→5→4→6。

22. 对于如图 8.19 所示的带权有向图，采用 Dijkstra 算法求从顶点 0 到顶点 6 的最短路径，path[1..6]的元素依次是________。

答：0,1,2,5,3,4。

23. Dijkstra 算法从源点到其余各顶点的最短路径的路径长度按____①____次序依次产生，该算法在边上的权出现____②____情况时不能正确产生最短路径。

答：① 递增　② 负值。

24. 对于含有 n 个顶点、e 条边的带权图，采用 Floyd 算法求所有两个顶点之间的最短路径，在求出所有最短路径后 path[i][j]的元素表示________。

答：从顶点 i 到顶点 j 的最短路径上顶点 j 的前一个顶点的顶点编号。

25. 对于含有 n 个顶点、e 条边的带权图，采用 Floyd 算法求所有两个顶点之间的最短路径，$A_k[i][j]=\infty$，表示________。

答：考虑编号不大于 k 的所有顶点后，顶点 i 到顶点 j 之间没有路径。

26. 可以进行拓扑排序的有向图一定是________。

答：无环图。

27. 对于含有 n 个顶点、e 条边的有向无环图，拓扑排序算法的时间复杂度是________。

答：$O(n+e)$。

28. 一个含有 n 个顶点的有向图仅有唯一的拓扑序列，则该图的边数为________。

答：$n-1$。该有向图仅有唯一的拓扑序列，则只有唯一入度为 0 的顶点，其余所有顶点的入度为 1，则入度之和为 $n-1$，出度之和也为 $n-1$，所以有顶点度之和为 $2(n-1)$，而边数 $e=2(n-1)/2=n-1$。

29. AOE 网中从源点到汇点长度最长的路径称关键路径，该路径上的活动称为________。

答：关键活动。

30. 在一个 AOE 网中，某活动 a 的最早开始时间为 $e(a)$，最迟开始时间为 $l(a)$，该活动需要 c 天完成，若满足________，则称 a 为关键活动。

答：$e(a)=l(a)$。

8.3.3 判断题

1. 判断以下叙述的正确性。

(1) n 个顶点的无向图最多有 $n(n-1)$ 条边。

(2) 在有向图中，各顶点的入度之和等于各顶点的出度之和。

(3) 无论是有向图还是无向图，其邻接矩阵表示都是唯一的。

(4) 对同一个有向图来说，只保存出边的邻接表中结点的数目总是和只保存入边的邻接表中结点的数目一样多。

(5) 如果表示图的邻接矩阵是对称矩阵，则该图一定是无向图。

(6) 如果表示有向图的邻接矩阵是对称矩阵，则该有向图一定是完全有向图。

(7) 连通图的生成树包含了图中的所有顶点。

(8) 对 n 个顶点的连通图 G 来说，如果其中的某个子图有 n 个顶点、$n-1$ 条边，则该子图一定是 G 的生成树。

(9) 最小生成树是指边数最少的生成树。

(10) 从 n 个顶点的连通图中选取 $n-1$ 条权值最小的边即可构成最小生成树。

答：(1) 错误。n 个顶点的无向图最多有 $n(n-1)/2$ 条边。

(2) 正确。

(3) 正确。

(4) 正确。

(5) 错误。如完全有向图的邻接矩阵也是对称矩阵。

(6) 错误。

(7) 正确。

(8) 错误。这样的子图不一定是连通图，而连通图的生成树一定是连通的。

(9) 错误。图的所有生成树的边数是相同的，其中权值之和最小的生成树为最小生成树。

(10) 错误。要求最小生成树中不能有回路。

2. 判断以下叙述的正确性。

(1) 强连通图不能进行拓扑排序。

(2) 只要带权无向图中没有权值相同的边,其最小生成树就是唯一的。

(3) 只要带权无向图存在权值相同的边,其最小生成树就不可能是唯一的。

(4) 关键路径是由权值最大的边构成的。

(5) 一个 AOE 网可能有多条关键路径,这些关键路径的长度可以不相同。

(6) 求单源最短路径的 Dijkstra 算法不适用于有回路的带权有向图。

(7) 求单源最短路径的 Dijkstra 算法不适用于有负权边的带权有向图。

(8) 最短路径一定是简单路径。

答:(1) 正确。强连通图中一定存在回路,所以不能进行拓扑排序。

(2) 正确。

(3) 错误。不一定,如一个无向图有 3 条边,权分别为 1、1、2,其最小生成树是唯一的。

(4) 错误。

(5) 错误。所有关键路径的长度是相同的。

(6) 错误。

(7) 正确。

(8) 正确。

3. 判断以下叙述的正确性。

(1) 连通分量是无向图中的极小连通子图。

(2) 强连通分量是有向图中的极大强连通子图。

(3) 在一个有向图的拓扑序列中若顶点 a 在顶点 b 之前,则图中必有一条边 $<a,b>$。

(4) 对于有向图 G,如果从任一顶点出发进行一次深度优先或广度优先遍历能访问到每个顶点,则该图一定是完全图。

(5) 在连通图的广度优先遍历中一般要采用队列来暂存刚访问过的顶点。

(6) 在图的深度优先遍历中一般要采用栈来暂存刚访问过的顶点。

(7) 广度优先遍历方法仅仅适合无向图的遍历而不适合有向图的遍历。

答:(1) 错误。连通分量是无向图中的极大连通子图。

(2) 正确。

(3) 错误。拓扑序列中顶点 a 在顶点 b 之前表示 a 到 b 有路径或者有边。

(4) 错误。如果有向图构成一个有向环,则从任一顶点出发均能访问到每个顶点,但该图却非完全图。

(5) 正确。

(6) 正确。深度优先遍历递归算法的执行就是用栈来暂存刚访问过的顶点。

(7) 错误。广度优先遍历方法既适合无向图的遍历也适合有向图的遍历。

4. 判断以下叙述的正确性。

(1) 无环有向图才能进行拓扑排序。

(2) 拓扑排序算法不适合无向图的拓扑排序。

(3) 关键路径是 AOE 网中从源点到汇点的最长路径。

(4) 在表示某工程的 AOE 网中,加速其关键路径上的任意关键活动均可缩短整个工程

的完成时间。

(5) 在 AOE 图中，所有关键路径上共有的某个活动的时间缩短 C 天，整个工程的时间也必定缩短 C 天。

(6) 在 AOE 图中，延长关键活动的时间会导致延长整个工程的工期。

(7) 当一个 AOE 网中所有活动的时间都不相同时，其关键路径是唯一的。

答：(1) 正确。

(2) 正确。多个顶点构成的无向图中必然存在回路。

(3) 正确。

(4) 错误。一个 AOE 网中可能存在多条关键路径，只有缩短所有关键路径上共有的关键活动才有可能缩短整个工程的完成时间。

(5) 错误。不一定，当 C 取值不当时，AOE 中关键路径可能发生改变。

(6) 正确。

(7) 错误。这样的 AOE 网的关键路径不一定唯一。

8.3.4 简答题

1. 简述图有哪两种主要的存储结构，并说明各种存储结构在图中的不同运算(如图的遍历、求最小生成树、最短路径和拓扑排序等)中有什么样的优越性？

答：图的存储结构主要有邻接矩阵和邻接表。

(1) 邻接矩阵：容易判定图中任意两个顶点之间是否有边相连，所以对于图的遍历是可行的。同时特别方便提取一条边的权值，所以在求最小生成树和最短路径时采用邻接矩阵作为存储结构。

(2) 邻接表：容易找到任一顶点的所有邻接点，所以邻接表对于图的遍历也是可行的，并且方便拓扑排序。但要判断任意两个顶点 i 和 j 之间是否有边相连，则需搜索第 i 个及第 j 个单链表，这一点不如邻接矩阵方便。

2. 回答以下关于图的问题：

(1) 有 n 个顶点的强连通图最多需要多少条边？最少需要多少条边？

(2) 表示一个有 1000 个顶点、1000 条边的有向图的邻接矩阵有多少个矩阵元素？

(3) 对于一个有向图，不用拓扑排序，如何判断图中是否存在环？

答：(1) 有 n 个顶点的强连通图最多有 $n(n-1)$ 条边(构成一个完全有向图的情况)；最少有 n 条边(n 个顶点依次首尾相接构成一个环的情况)。

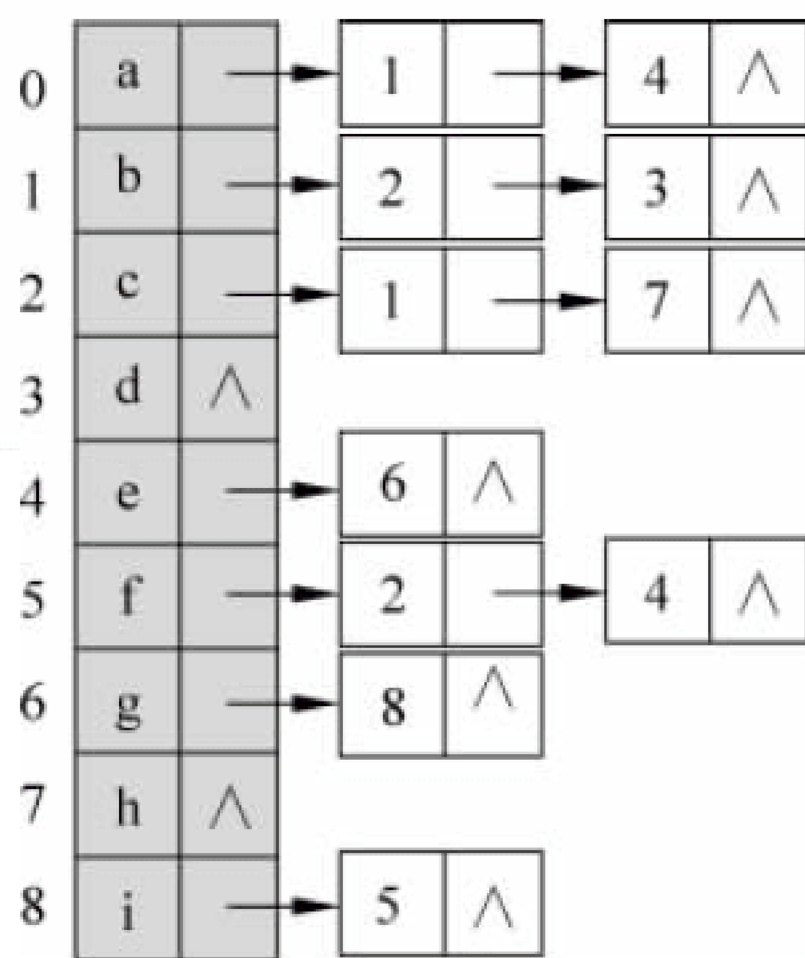


图 8.20 一个邻接表

(2) 这样的矩阵共有 1000^2 个矩阵元素。

(3) 对于有向图进行深度优先遍历。如果从有向图中某个顶点 v 出发的遍历 DFS(v)，在访问顶点 u 后，又出现一条从顶点 u 到顶点 v 的回边，由于顶点 u 在生成树上是顶点 v 的子孙，则有向图中必定存在包含顶点 v 到顶点 u 的环。

3. 一个有向图 G 的邻接表存储如图 8.20 所示，给出该图的所有强连通分量。

答：由邻接表得到的图 G 如图 8.21 所示。图 G 中顶点 e、f、g、i 构成一个环，其他顶点无法加入形成双向路径，顶点 b、c 构成一个环，其他顶点无法加入形成双向路径。所以该

图的所有强连通分量如图 8.22 所示。

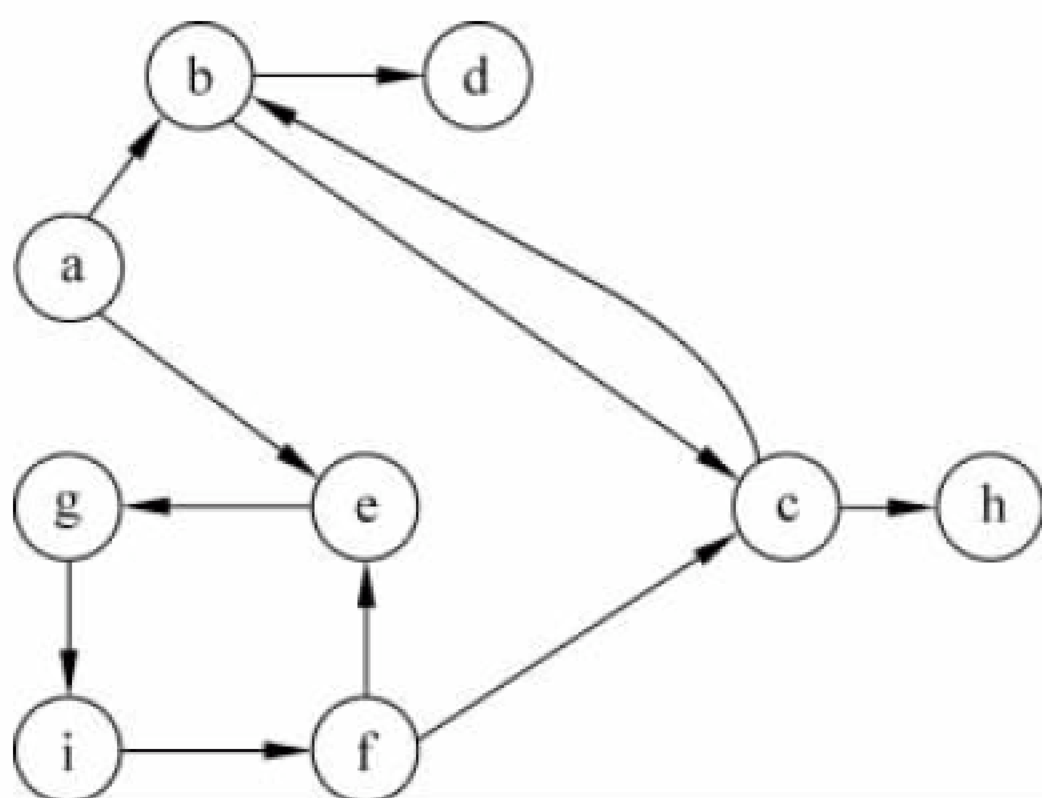


图 8.21 一个有向图

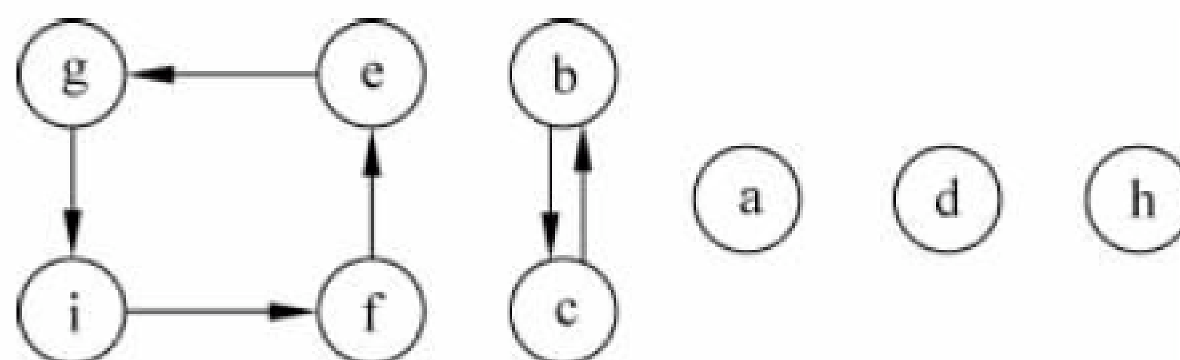


图 8.22 强连通分量

4. 有一个带权有向图如图 8.23 所示，回答以下问题：

(1) 给出该图的邻接矩阵表示。

(2) 给出该图的邻接表表示。

(3) 给出该图的逆邻接表表示。

(4) 和邻接表相比，逆邻接表的主要作用是什么？

答：(1) 该图的邻接矩阵表示如下。

$$
\begin{bmatrix}
0 & 2 & 5 & 3 & \infty & \infty & \infty \\
\infty & 0 & 2 & \infty & \infty & 8 & \infty \\
\infty & \infty & 0 & 1 & 3 & 5 & \infty \\
\infty & \infty & \infty & 0 & 5 & \infty & \infty \\
\infty & \infty & \infty & \infty & 0 & 3 & 9 \\
\infty & \infty & \infty & \infty & \infty & 0 & 5 \\
\infty & \infty & \infty & \infty & \infty & \infty & 0
\end{bmatrix}
$$

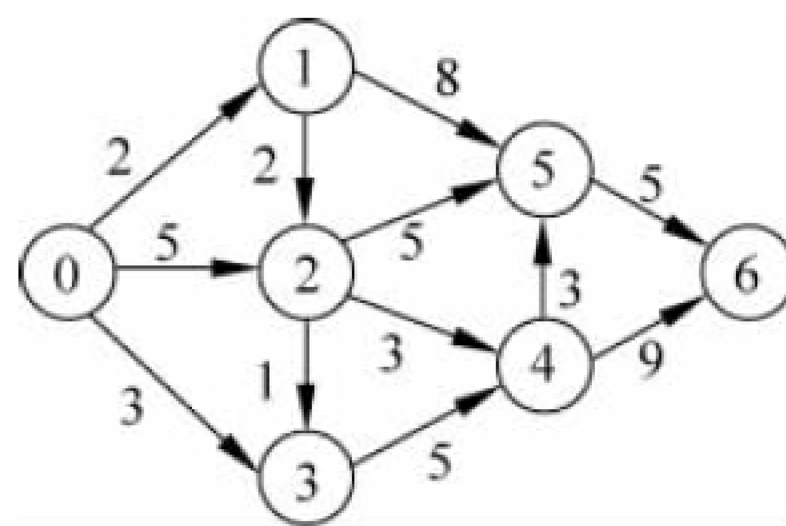


图 8.23 一个带权有向图

(2) 该图的邻接表表示如图 8.24 所示。

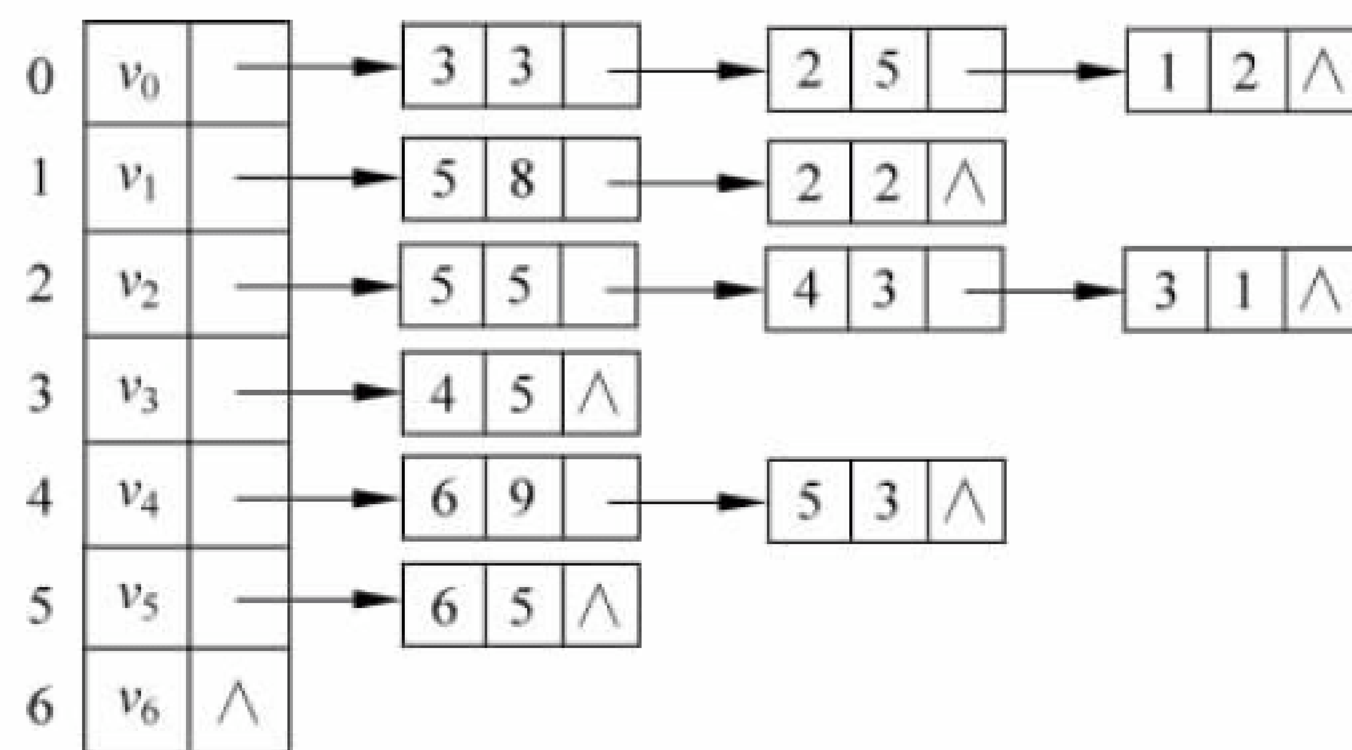


图 8.24 一个邻接表

(3) 该图的逆邻接表表示如图 8.25 所示。

(4) 和邻接表相比，逆邻接表更适合于求顶点的入度，找顶点的入边和入边邻接点。

5. 证明当深度优先遍历算法应用于一个连通图时遍历过程中所经历的边形成一棵树。

证明：由深度优先遍历算法可知，一个连通图中的每个顶点访问一次并且仅访问一次。在遍历过程中，从一个顶点到另外一个顶点时必须经过连接这两个顶点的边。这样，当深度

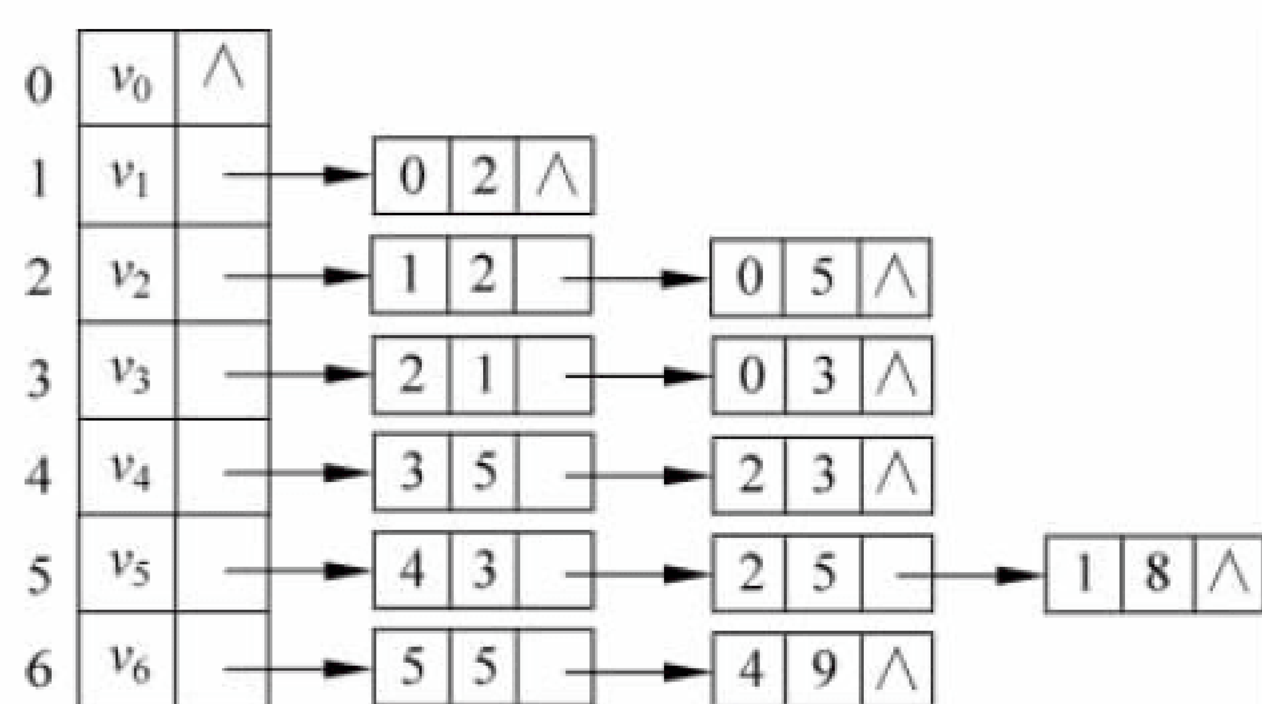


图 8.25　一个逆邻接表

优先遍历将图中的全部顶点都访问一次后共经过了其中 $n-1$ 条边，而这 $n-1$ 条边恰好把图中的 n 个顶点全部连通，即图的 n 个顶点和这 $n-1$ 条边构成了图的一个连通分量。具有 n 个顶点和 $n-1$ 条边的连通图为树。

6. 有如图 8.26 所示的带权有向图 G，试回答以下问题。

(1) 给出从顶点 1 出发的一个深度优先遍历序列和一个广度优先遍历序列。

(2) 给出 G 的一个拓扑序列。

(3) 给出从顶点 1 到顶点 8 的最短路径和关键路径。

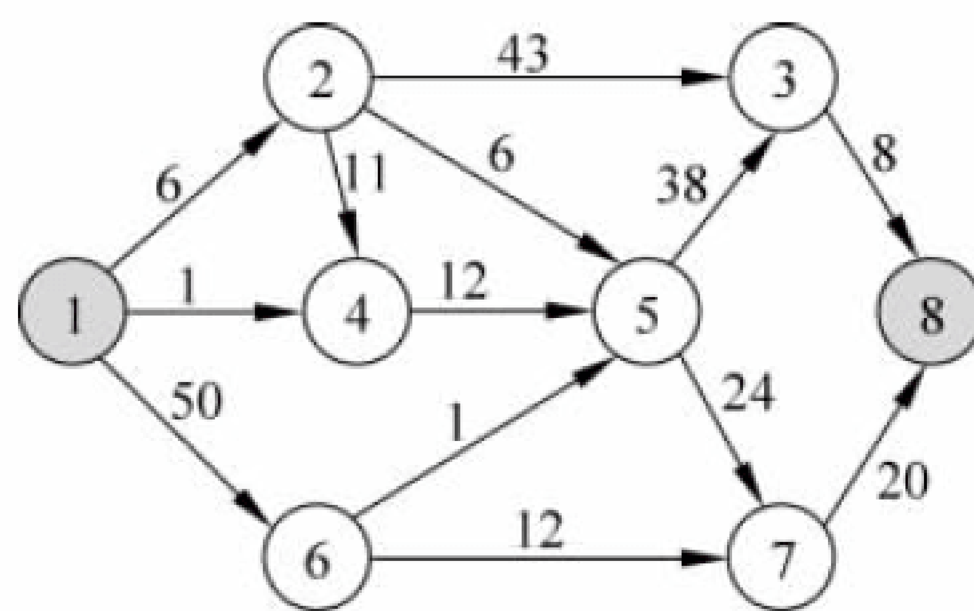


图 8.26　一个带权有向图 G

答：(1) 从顶点 1 出发的一个深度优先遍历序列为 1,2,3,8,4,5,7,6。从顶点 1 出发的一个广度优先遍历序列为 1,2,6,4,3,5,7,8。

(2) G 的一个拓扑序列是 1,2,4,6,5,3,7,8(或 1,6,2,4,5,3,7,8 等)。

(3) 从顶点 1 到顶点 8 的最短路径为 1,2,5,7,8(路径长度为 56)，关键路径(即为最长的路径)为 1,6,5,3,8(路径长度为 97)。

7. 有一个带权无向图，其邻阵矩阵数组表示如下：

$$
\begin{matrix}0\\1\\2\\3\\4\\5\\6\\7\end{matrix}
\begin{bmatrix}
0 & 3 & 5 & \infty & \infty & \infty & 9 & \infty \\
3 & 0 & 6 & \infty & \infty & \infty & \infty & 10 \\
5 & 6 & 0 & \infty & \infty & \infty & \infty & 4 \\
\infty & \infty & \infty & 0 & 3 & 6 & \infty & \infty \\
\infty & \infty & \infty & 3 & 0 & 5 & \infty & \infty \\
\infty & \infty & \infty & 6 & 5 & 0 & \infty & \infty \\
9 & \infty & \infty & \infty & \infty & \infty & 0 & 7 \\
\infty & 10 & 4 & \infty & \infty & \infty & 7 & 0
\end{bmatrix}
$$

试完成下列要求：

(1) 画出该图的一种邻接表(图中顶点编号为 0～7)。

(2) 在给出的邻接表中从顶点 0 出发进行深度优先遍历，得到访问的顶点序列是什么？并据此判断该图是否为连通图。

(3) 如果该图是不连通的，分别从顶点 0 和顶点 3 出发，采用 Prim 算法构造最小生成树(森林)。

答：(1) 该图的带权邻接表如图 8.27 所示。

(2) 从顶点 1 出发得到的一个深度优先遍历序列为 0,1,2,7,6。由此可知该图为非连通图(没有访问全部的顶点)。

顶点	边结点 1	边结点 2	边结点 3
0	1, 3	2, 5	6, 9, ∧
1	0, 3	2, 6	7, 10, ∧
2	0, 5	1, 6	7, 4, ∧
3	4, 3	5, 6, ∧	
4	3, 3	5, 5, ∧	
5	3, 6	4, 5, ∧	
6	0, 9	7, 7, ∧	
7	1, 10	2, 4	6, 7, ∧

图 8.27 带权邻接表

(3) 该图是不连通的,有两个连通分量,分别从顶点 0 和顶点 3 出发采用 Prim 算法构造的最小生成树(森林)如图 8.28 所示。

8. 给定如图 8.29 所示的带权无向图 G。

(1) 画出该图的邻接表存储结构。

(2) 根据该图的邻接表存储结构,从顶点 0 出发,调用 DFS 和 BFS 算法遍历该图,给出相应的遍历序列。

(3) 给出采用 Kruskal 算法构造最小生成树的过程。

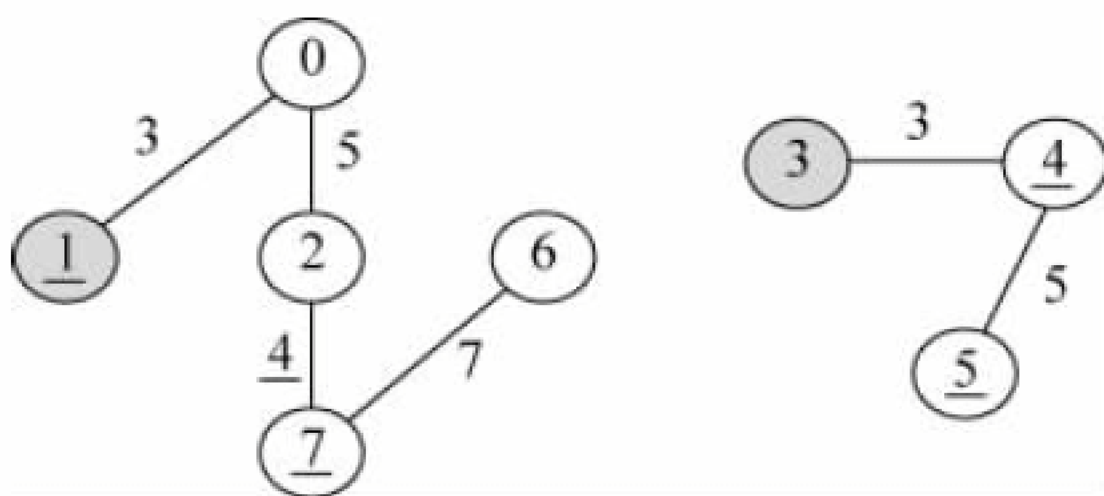


图 8.28 最小生成树(森林)

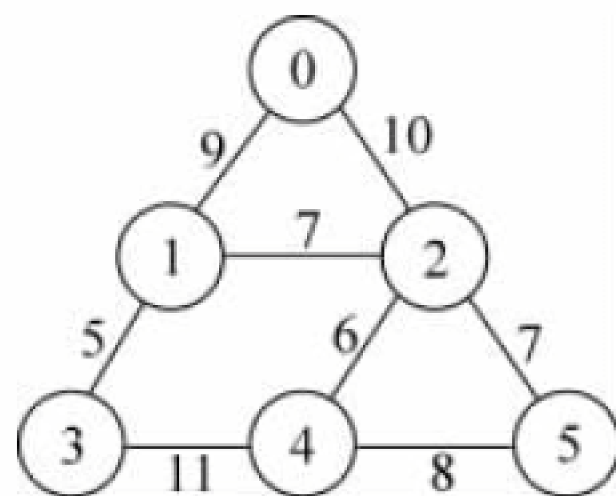


图 8.29 一个带权无向图

答：(1) 图 G 的邻接表存储结构如图 8.30 所示。

(2) 从顶点 0 出发,DFS 遍历序列为 0,1,2,4,3,5,BFS 遍历序列为 0,1,2,3,4,5。

顶点	边结点 1	边结点 2	边结点 3	边结点 4
0	1, 9	2, 10, ∧		
1	0, 9	2, 7	3, 5, ∧	
2	0, 10	1, 7	4, 6	5, 7, ∧
3	1, 5	4, 11, ∧		
4	2, 6	3, 11	5, 8, ∧	
5	2, 7	4, 8, ∧		

图 8.30 图 G 的邻接表

（3）采用 Kruskal 算法构造最小生成树的过程如图 8.31 所示。

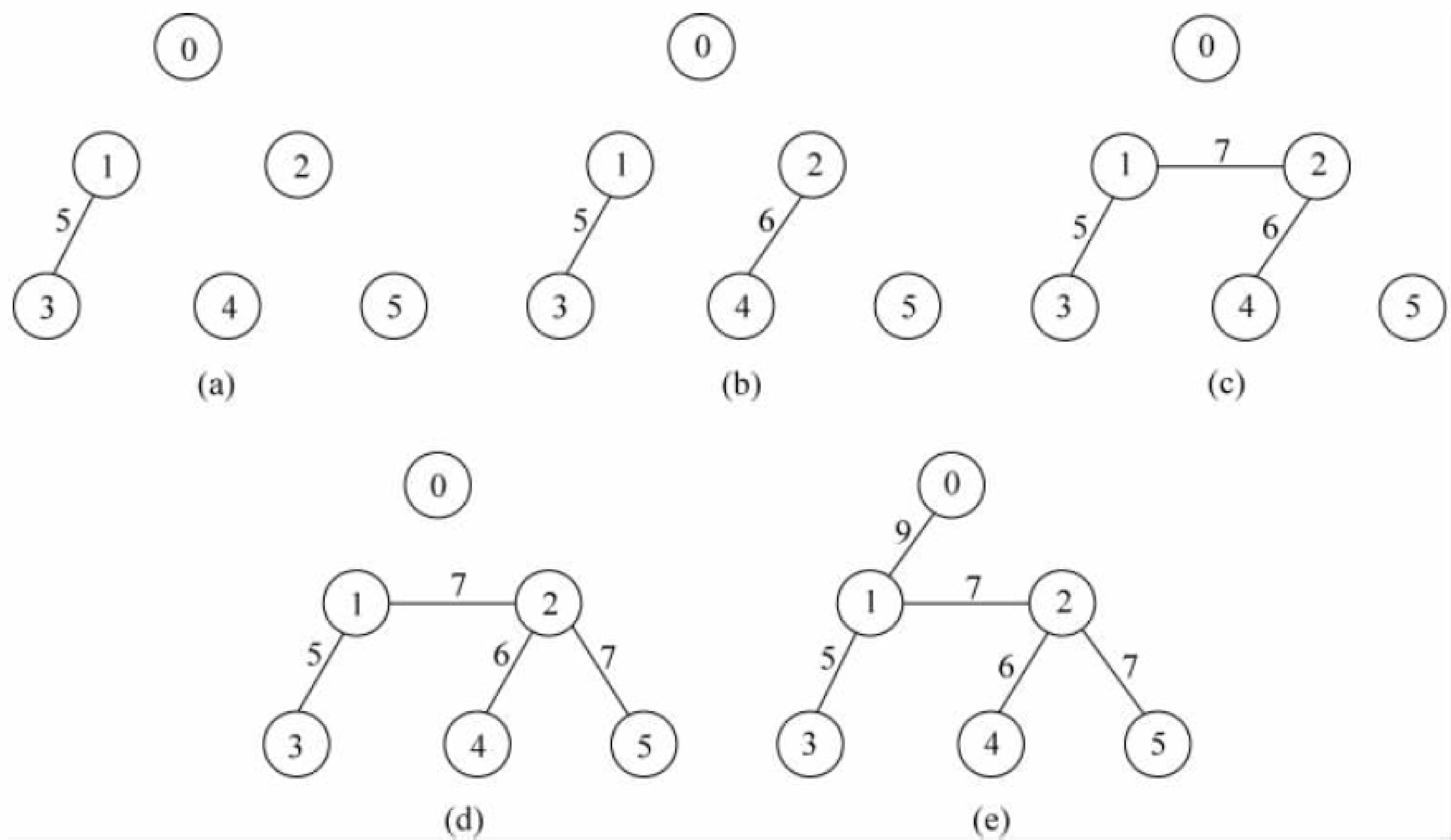


图 8.31　Kruskal 算法构造最小生成树的过程

9. 对于图 8.32 所示的有向网，试利用 Dijkstra 算法求出从源点 1 到其他各顶点的最短路径，并写出执行过程。

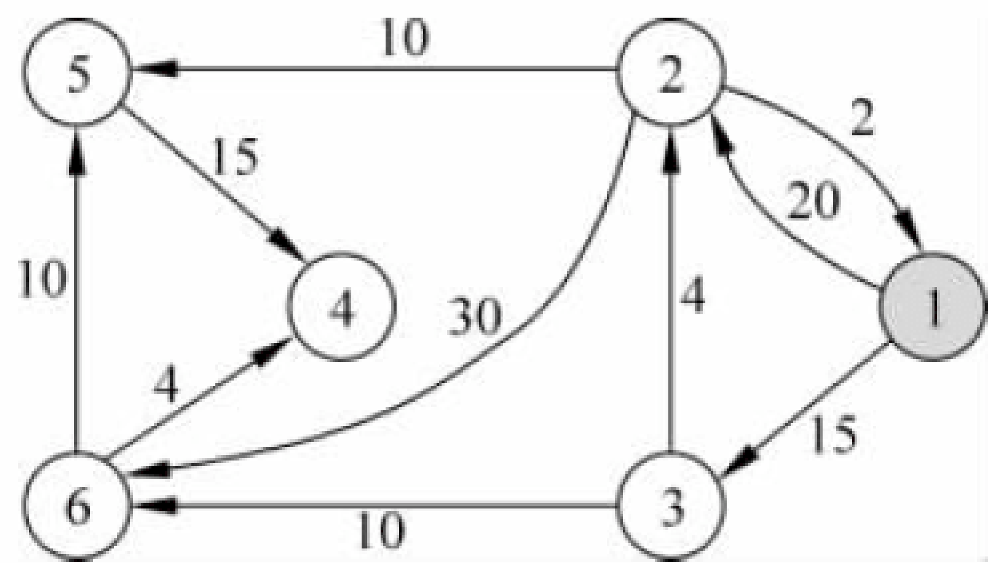


图 8.32　带权有向图

答：求解过程如下。

S	dist[1]	dist[2]	dist[3]	dist[4]	dist[5]	dist[6]	path[1]	path[2]	path[3]	path[4]	path[5]	path[6]
{1}	0	20	15	∞	∞	∞	1	1	1	0	0	0
{1 3}	0	19	15	∞	∞	25	1	3	1	0	0	3
{1 2 3}	0	19	15	∞	29	25	1	3	1	0	2	3
{1 2 3 6}	0	19	15	29	29	25	1	3	1	6	2	3
{1 2 3 5 6}	0	19	15	29	29	25	1	3	1	6	2	3
{1 2 3 4 5 6}	0	19	15	29	29	25	1	3	1	6		3

所以有：

从顶点 1 到顶点 2 的最短距离为 19，路径为 1，3，2；

从顶点 1 到顶点 3 的最短距离为 15，路径为 1，3；

从顶点 1 到顶点 4 的最短距离为 29，路径为 1，3，6，4；

从顶点 1 到顶点 5 的最短距离为 29，路径为 1，3，2，5

从顶点 1 到顶点 6 的最短距离为 25，路径为 1，3，6。

10. 带权图(权值非负，表示边连接的两顶点间的距离)的最短路径问题是找出从初始顶点到目标顶点之间的一条最短路径。假设从初始顶点到目标顶点之间存在路径，现有一种解决该问题的方法，具体的解题步骤如下：

① 设最短路径初始时仅包含初始顶点，令当前顶点 u 为初始顶点。

② 选择离 u 最近且尚未在最短路径中的一个顶点 v，加入到最短路径中，修改当前顶点 $u=v$。

③ 重复步骤②直到 u 是目标顶点时为止。

请问上述方法能否求得最短路径？若该方法可行，请证明；否则，请举例说明。

答：该方法不一定能(或不能)求得最短路径。

例如，在图 8.33(a)中，设初始顶点为 1、目标顶点为 4，欲求从顶点 1 到顶点 4 之间的最短路径。显然这两顶点之间的最短路径长度为 2。但利用给定的方法求得的路径长度为 3，这条路径并不是这两个顶点之间的最短路径。

在图 8.33(b)中，设初始顶点为 1、目标顶点为 3，欲求从顶点 1 到顶点 3 之间的最短路径。利用给定的方法无法求出顶点 1 到顶点 3 的路径。

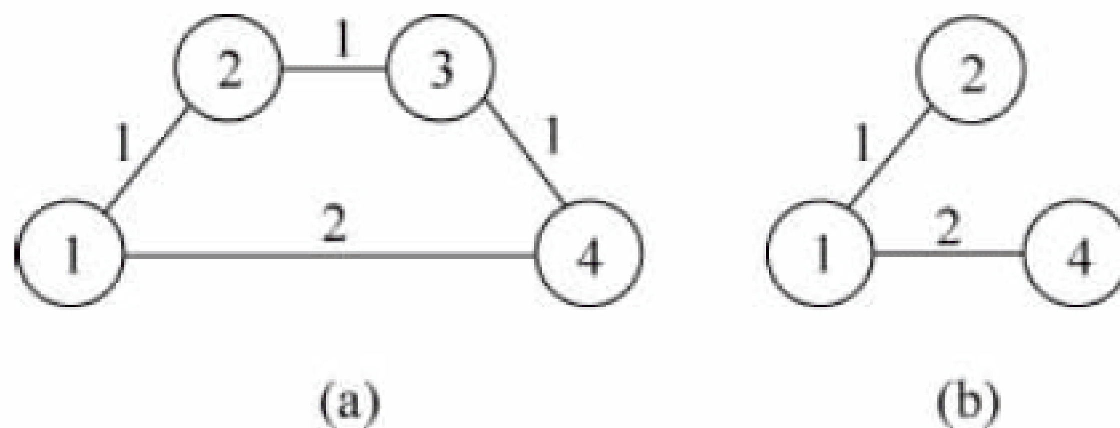


图 8.33 求最短路径的反例

11. 设 **A** 为一个不带权图的 0/1 邻接矩阵，定义如下：

$$\mathbf{A}^{(1)} = \mathbf{A}$$
$$\mathbf{A}^{(m)} = \mathbf{A}^{(m-1)} \times \mathbf{A}$$

试证明 $A^{(m)}[i][j]$的值即为从顶点 i 到顶点 j 的路径长度为 m 的路径条数。

证明：采用数学归纳法求证。

当 $m=1$ 时，即 $\mathbf{A}^{(1)}$ 为邻接矩阵 **A**，而其中 $A[i][j]$的值只能是 0 或 1。若 $A[i][j]=0$，则说明图中没有从顶点 i 到顶点 j 的路径，即对应的边数为 0；若 $A[i][j]=1$，则说明图中存在一条从顶点 i 到顶点 j 的路径，即对应的边数为 1，此时结论成立。

假设 $m=k$ 时结论成立，即 $A^{(k)}[i][j]$的值为从顶点 i 到顶点 j 的路径长度为 k 的路径条数。

当 $m=k+1$ 时，由于 $A^{(k+1)}[i][j] = \sum_{l=0}^{n-1} A^{(k)}[i][l] \times A[l][j]$(设 n 为图中的顶点数)，其中 $A^{(k)}[i][l]$是从顶点 i 到顶点 l 的路径长度为 k 的数目，$A[l][j]$是从顶点 l 到顶点 j 的路径长度为 1 的数目。那么，对于任何一个 l，$A^{(k)}[i][l] \times A[l][j]$即为从顶点 i 到达顶点 l 后再直接到达 j 的路径长度为 $k+1$ 的数目，因此，对于所有的顶点 $l(0 \leqslant l < n)$，$A^{(k+1)}[i][j] = \sum_{l=0}^{n-1} A^{(k)}[i][l] \times A[l][j]$ 即为从顶点 i 到顶点 j 的路径长度为 $k+1$ 的路径条数。

12. 对于无向图 8.34，按照 Floyd 算法，给出所有两个顶点之间的最短路径和最短路径长度。

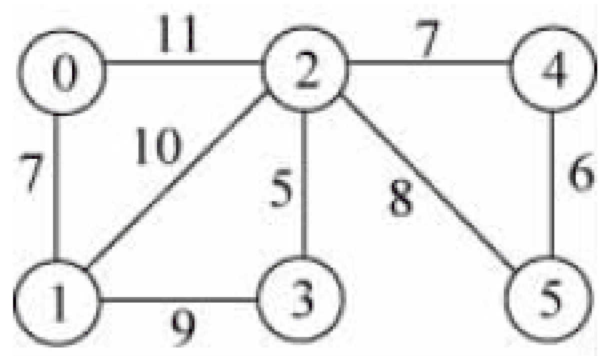


图 8.34 一个无向图

答：求解过程如下。

A(0)

0	1	2	3	4	5
0	7	11	∞	∞	∞
7	0	10	9	∞	∞
11	10	0	5	7	8
∞	9	5	0	∞	∞
∞	∞	7	∞	0	6
∞	∞	8	∞	6	0

path(0)

0	1	2	3	4	5
-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1

A(1)

0	1	2	3	4	5
0	7	11	**16**	∞	∞
7	0	10	9	∞	∞
11	10	0	5	7	8
16	9	5	0	∞	∞
∞	∞	7	∞	0	6
∞	∞	8	∞	6	0

path(1)

0	1	2	3	4	5
-1	-1	-1	**1**	-1	-1
-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1
1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1

A(2)

0	1	2	3	4	5
0	7	11	16	**18**	**19**
7	0	10	9	**17**	**18**
11	10	0	5	7	8
16	9	5	0	**12**	**13**
18	**17**	7	**12**	0	6
19	**18**	8	**13**	6	0

path(2)

0	1	2	3	4	5
-1	-1	-1	1	**2**	**2**
-1	-1	-1	-1	**2**	**2**
-1	-1	-1	-1	-1	-1
1	-1	-1	-1	**2**	**2**
2	**2**	-1	**2**	-1	-1
2	**2**	-1	**2**	-1	-1

A(3)

0	1	2	3	4	5
0	7	11	16	18	19
7	0	10	9	17	18
11	10	0	5	7	8
16	9	5	0	12	13
18	17	7	12	0	6
19	18	8	13	6	0

path(3)

0	1	2	3	4	5
-1	-1	-1	1	2	2
-1	-1	-1	-1	2	2
-1	-1	-1	-1	-1	-1
1	-1	-1	-1	2	2
2	2	-1	2	-1	-1
2	2	-1	2	-1	-1

A(4)

0	1	2	3	4	5
0	7	11	16	18	19
7	0	10	9	17	18
11	10	0	5	7	8
16	9	5	0	12	13
18	17	7	12	0	6
19	18	8	13	6	0

path(4)

0	1	2	3	4	5
-1	-1	-1	1	2	2
-1	-1	-1	-1	2	2
-1	-1	-1	-1	-1	-1
1	-1	-1	-1	2	2
2	2	-1	2	-1	-1
2	2	-1	2	-1	-1

A(5)

0	1	2	3	4	5
0	7	11	16	18	19
7	0	10	9	17	18
11	10	0	5	7	8
16	9	5	0	12	13
18	17	7	12	0	6
19	18	8	13	6	0

path(5)

0	1	2	3	4	5
-1	-1	-1	1	2	2
-1	-1	-1	-1	2	2
-1	-1	-1	-1	-1	-1
1	-1	-1	-1	2	2
2	2	-1	2	-1	-1
2	2	-1	2	-1	-1

求解结果如下：

```
从 0 到 1 =>路径长度:7      路径:0,1
从 0 到 2 =>路径长度:11     路径:0,2
从 0 到 3 =>路径长度:16     路径:0,1,3
从 0 到 4 =>路径长度:18     路径:0,2,4
从 0 到 5 =>路径长度:19     路径:0,2,5
从 1 到 0 =>路径长度:7      路径:1,0
从 1 到 2 =>路径长度:10     路径:1,2
从 1 到 3 =>路径长度:9      路径:1,3
从 1 到 4 =>路径长度:17     路径:1,2,4
从 1 到 5 =>路径长度:18     路径:1,2,5
从 2 到 0 =>路径长度:11     路径:2,0
从 2 到 1 =>路径长度:10     路径:2,1
从 2 到 3 =>路径长度:5      路径:2,3
从 2 到 4 =>路径长度:7      路径:2,4
从 2 到 5 =>路径长度:8      路径:2,5
从 3 到 0 =>路径长度:16     路径:3,1,0
从 3 到 1 =>路径长度:9      路径:3,1
从 3 到 2 =>路径长度:5      路径:3,2
从 3 到 4 =>路径长度:12     路径:3,2,4
从 3 到 5 =>路径长度:13     路径:3,2,5
从 4 到 0 =>路径长度:18     路径:4,2,0
从 4 到 1 =>路径长度:17     路径:4,2,1
从 4 到 2 =>路径长度:7      路径:4,2
从 4 到 3 =>路径长度:12     路径:4,2,3
从 4 到 5 =>路径长度:6      路径:4,5
从 5 到 0 =>路径长度:19     路径:5,2,0
从 5 到 1 =>路径长度:18     路径:5,2,1
从 5 到 2 =>路径长度:8      路径:5,2
从 5 到 3 =>路径长度:13     路径:5,2,3
从 5 到 4 =>路径长度:6      路径:5,4
```

13. 设图 8.35 中的顶点表示村庄，有向边代表交通路线，若要建立一家医院，试问建在哪一个村庄能使各村庄的总体交通代价最少。

答：利用 Floyd 算法可求两顶点之间的最短路径长度。其邻接矩阵如下：

$$\boldsymbol{A}=\begin{bmatrix} 0 & 13 & \infty & 4 & \infty \\ 13 & 0 & 15 & \infty & 5 \\ \infty & \infty & 0 & 12 & \infty \\ 4 & \infty & 12 & 0 & \infty \\ \infty & \infty & 6 & 3 & 0 \end{bmatrix}$$

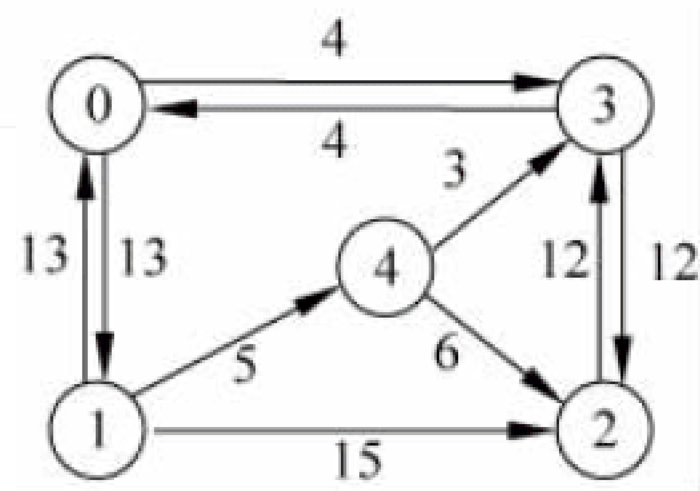


图 8.35　一个有向图 G

最后求得：

$$A_4 = \begin{bmatrix} 0 & 13 & 16 & 4 & 18 \\ 12 & 0 & 11 & 8 & 5 \\ 16 & 29 & 0 & 12 & 34 \\ 4 & 17 & 12 & 0 & 22 \\ 7 & 20 & 6 & 3 & 0 \end{bmatrix}$$

从 A_4 中求得每对村庄之间的最少交通代价。假设医院建在 i 村庄时，其他各村庄往返的总体交通代价如表 8.1 所示，显然把医院建在村庄 3 时总体交通代价最少。

表 8.1　交通代价表

医院建在的村庄	各村庄往返的总体交通代价
0	12+16+4+7+13+16+4+18=90
1	13+29+17+20+12+11+8+5=115
2	16+11+12+6+16+29+12+34=136
3	4+8+12+3+4+17+12+22=82
4	18+5+34+22+7+20+6+3+0=115

14. 对于有向无环图：

(1) 叙述求拓扑有序序列的步骤。

(2) 对于如图 8.36 所示的图 G，写出它的 4 个不同的拓扑序列。

答：(1) 拓扑排序的基本步骤如下。

① 从图中选择一个没有前驱(即入度为 0)的顶点并输出它。

② 从图中删去该顶点，并且删去从该顶点发出的全部有向边。

③ 重复上述两步，直到剩余的图中不再存在没有前驱的顶点为止。

(2) 它的 4 个不同的拓扑序列是 12345678、12354678、12347856、12347568(实际上不止 4 个拓扑序列，这里只给出了 4 个)。

15. 设有向图 G 如图 8.37 所示。

(1) 写出所有的拓扑序列。

(2) 在添加一条边后只有唯一的拓扑序列，问应该添加哪条边？

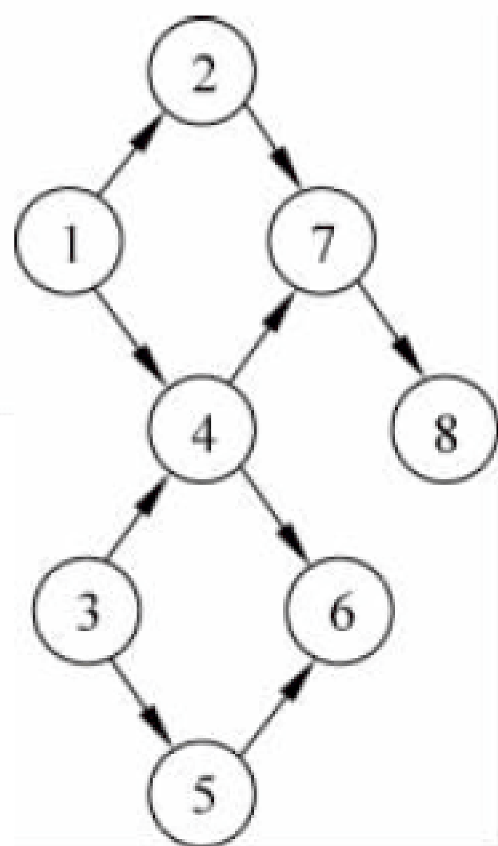


图 8.36　一个有向图

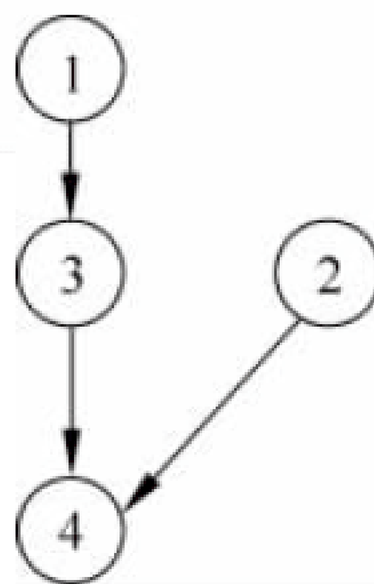


图 8.37　一个有向图

答：(1) 所有的拓扑序列为 1324、1234、2134。

(2) 在添加一条边<3,2>后(使入度为 0 的顶点和出度为 0 的顶点都是唯一的)如图 8.38 所示，仅有唯一的拓扑序列 1324。

16. 对于如图 8.39 所示的 AOE 网，求：

(1) 每项活动 a_i 的最早开始时间 $e(a_i)$和最迟开始时间 $l(a_i)$。

(2) 完成此工程最少需要多少天(设边上权值为天数)?

(3) 哪些是关键活动?

(4) 是否存在某项活动，当其提高速度后能使整个工程缩短工期?

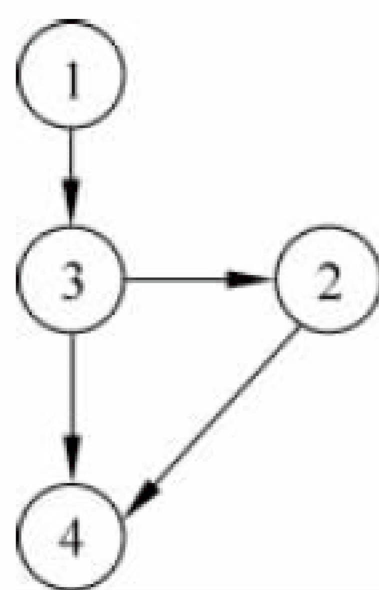


图 8.38　改动后的有向图

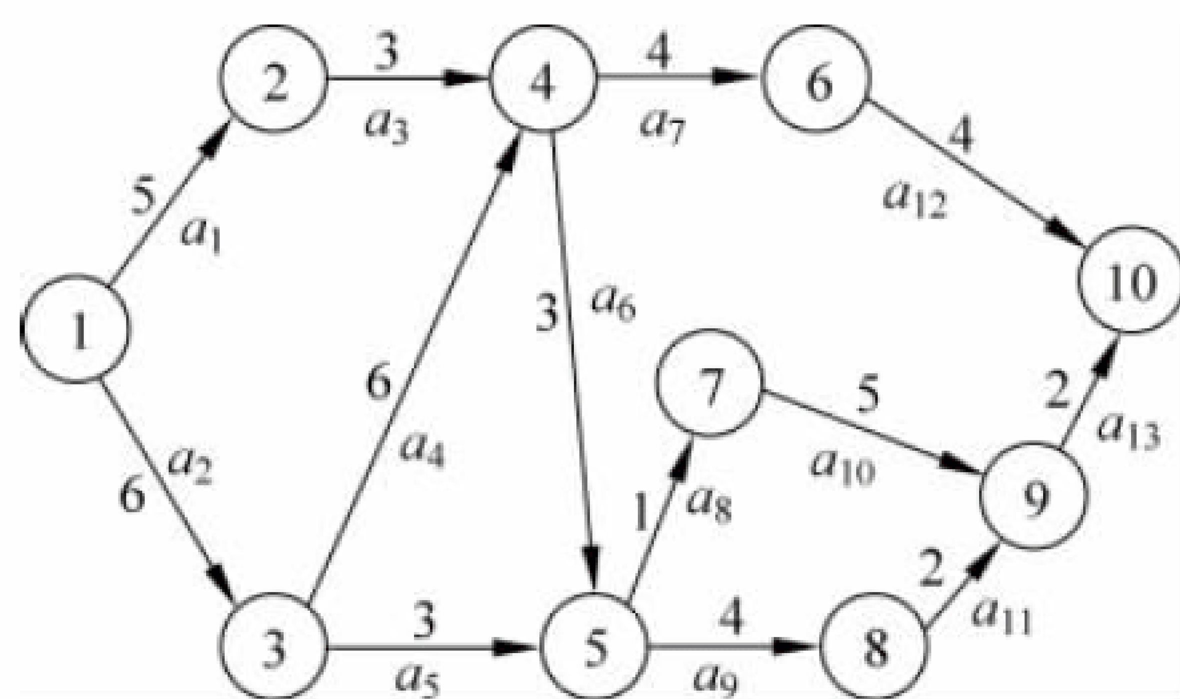


图 8.39　一个 AOE 网

答：(1)该 AOE 网的一个拓扑序列为 1～10，按该拓扑序列顺序求所有事件的最早发生时间如下。

ve(1)=0；　ve(2)=5；

ve(3)=6；　ve(4)=MAX{ve(2)+3,ve(3)+6}=12；

ve(5)=MAX{ve(3)+3,ve(4)+3}=15；　ve(6)=ve(4)+4=16；

ve(7)=ve(5)+1=16；　ve(8)=ve(5)+4=19；

ve(9)=MAX{ve(7)+5,ve(8)+2}=21；　ve(10)=MAX{ve(6)+4,ve(9)+2}=23。

该 AOE 网的一个逆拓扑序列为 10～1，按该逆拓扑序列顺序求所有事件的最迟发生时间如下。

vl(10)=ve(10)=23；　vl(9)=vl(10)−2=21；

vl(8)=vl(9)−2=19；　vl(7)=vl(9)−5=16；

vl(6)=vl(10)−4=19；　vl(5)=MIN{vl(7)−1,vl(8)−4}=15；

vl(4)=MIN{vl(6)−4,vl(5)−3}=12；　vl(3)=MIN{vl(4)−6,vl(5)−3}=6；

vl(2)=vl(4)−3=9；　vl(1)=MIN{vl(2)−5,vl(3)−6}=0。

求所有活动的 $e()$、$l()$和 $d()$如下。

活动 a_1：$e(a_1)$=ve(1)=0　$l(a_1)$=vl(2)−5=4　$d(a_1)=4$

活动 a_2：$e(a_2)$=ve(1)=0　$l(a_2)$=vl(3)−6=0　$d(a_2)=0$

活动 a_3：$e(a_3)$=ve(2)=5　$l(a_3)$=vl(4)−3=8　$d(a_3)=3$

活动 a_4：$e(a_4)$=ve(2)=6　$l(a_4)$=vl(4)−6=6　$d(a_4)=0$

活动 a_5：$e(a_5)$=ve(3)=6　$l(a_5)$=vl(5)−3=12　$d(a_5)=6$

活动 a_6：$e(a_6)$=ve(3)=12　$l(a_6)$=vl(5)−3=12　$d(a_6)=0$

活动 a_7：$e(a_7)$=ve(4)=12　$l(a_7)$=vl(6)−4=15　$d(a_7)=3$

活动 a_8：$e(a_8)=\mathrm{ve}(5)=15$　　$l(a_8)=\mathrm{vl}(7)-1=15$　　$d(a_8)=0$
活动 a_9：$e(a_9)=\mathrm{ve}(5)=15$　　$l(a_9)=\mathrm{vl}(8)-4=15$　　$d(a_9)=0$
活动 a_{10}：$e(a_{10})=\mathrm{ve}(6)=16$　　$l(a_{10})=\mathrm{vl}(9)-5=16$　　$d(a_{10})=0$
活动 a_{11}：$e(a_{11})=\mathrm{ve}(7)=19$　　$l(a_{11})=\mathrm{vl}(9)-2=19$　　$d(a_{11})=0$
活动 a_{12}：$e(a_{12})=\mathrm{ve}(8)=16$　　$l(a_{12})=\mathrm{vl}(10)-4=19$　　$d(a_{12})=3$
活动 a_{13}：$e(a_{13})=\mathrm{ve}(8)=21$　　$l(a_{13})=\mathrm{vl}(10)-2=21$　　$d(a_{13})=0$

(2) 完成此工程最少需要 23 天。

(3) 从以上计算得出，关键活动为 a_2、a_4、a_6、a_8、a_9、a_{10}、a_{11} 和 a_{13}。这些活动构成了两条关键路径，即 a_2、a_4、a_6、a_8、a_{10}、a_{13} 和 a_2、a_4、a_6、a_9、a_{11}、a_{13}。

(4) 存在 a_2、a_4、a_6、a_{13}活动，当其提高速度后能使整个工程缩短工期。

8.3.5 算法设计题

1.【图的邻接表运算算法】假设带权有向图 G 采用邻接表存储。设计一个算法增加一条边<i,j>，其权值为 w，假设顶点 i、j 已存在，原来图中不存在<i,j>边。

解：建立一个边结点 p，置 p->adjvex=j，p->weight=w。将 p 结点插入到 G->adjlist[i]单链表的开头。对应的算法如下：

```
void AddEdge(AdjGraph *&G,int i,int j,int w)
{   ArcNode *p;
    p=(ArcNode *)malloc(sizeof(ArcNode));
    p->adjvex=j;
    p->weight=w;
    p->nextarc=G->adjlist[i].firstarc;
    G->adjlist[i].firstarc=p;
    G->e++;
}
```

2.【图的邻接表运算算法】假设带权有向图 G 采用邻接表存储，假设顶点 i、j 已存在，设计一个算法删除一条已存在的边<i,j>。

解：让 pre 指向 G->adjlist[i]边单链表的首结点。如果 pre 为空，则返回；若 pre 为不空，并且 pre->adjvex==j，则删除并释放 pre 结点；否则用 pre、p 在该边单链表中查找到 adjvex 域为 j 的 p 结点，通过其前驱结点 pre 删除并释放 p 结点。对应的算法如下：

```
void DelEdge(AdjGraph *&G,int i,int j)
{   ArcNode *pre,*p;
    pre=G->adjlist[i].firstarc;
    if (pre==NULL) return;
    if (pre->adjvex==j)
    {   G->adjlist[i].firstarc=pre->nextarc;
        free(pre);
        G->e--;
    }
    else
    {   p=pre->nextarc;
```

```
            while (p!= NULL && p->adjvex!= j)
            {   pre = p;
                p = p->nextarc;
            }
            pre->nextarc = p->nextarc;
            free(p);
            G->e--;
        }
    }
```

3.【图的邻接表运算算法】假设带权有向图 G 采用邻接表存储,设计一个算法输出顶点 i 的所有入边邻接点。

解:若顶点 i 错误,直接返回。用 j 扫描所有的单链表,对于 G->adjlist[j]单链表,如果其中存在 adjvex 域为 i 的结点,表示顶点 j 是顶点 i 的入边邻接点,则输出 j。对应的算法如下:

```
void AllInNeig(AdjGraph *G,int i)
{   int j;
    ArcNode *p;
    if (i<0 || i>G->n)
        return;
    for (j = 0;j<G->n;j++)
    {   p = G->adjlist[j].firstarc;
        while (p!= NULL)
        {   if (p->adjvex == i)
            {   printf("%d ",j);
                break;
            }
            p = p->nextarc;
        }
    }
    printf("\n");
}
```

4.【图的遍历算法】假设无向图采用邻接表存储,编写一个算法求连通分量的个数并输出各连通分量的顶点集。

解:以深度优先遍历来求图 G 的连通分量的个数。对应的算法如下:

```
int DFSTrave(AdjGraph *G)
{   int k,num = 0;                  //num 记录连通分量的个数
    for (k = 0;k<G->n;k++)
        visited[k] = 0;
    for (k = 0;k<G->n;k++)
        if (visited[k] == 0)
        {   num++;
            printf("第%d个连通分量顶点集:",num);
            DFS(G,k);               //DFS 是《教程》中的深度优先遍历算法
```

```
            printf("\n");
        }
    return num;
}
```

说明：本题也可采用广度优先遍历算法。

5.【图的遍历算法】假设图采用邻接表存储，分别写出基于 DFS 和 BFS 遍历的算法来判断顶点 i 和顶点 $j(i\neq j)$之间是否有路径。

解：先置全局变量 visited[ ]为 0，然后从顶点 i 开始进行某种遍历，遍历之后，若 visited[j]=0，说明顶点 i 与顶点 j 之间没有路径，否则说明它们之间存在路径。基于 DFS 遍历的算法如下：

```
int visited[MAXV];                              //全局变量数组
bool DFSTrave(AdjGraph *G,int i,int j)
{   int k;
    for (k=0;k<G->n;k++)
        visited[k]=0;
    DFS(G,i);                                   //从顶点 i 开始进行深度优先遍历
    if (visited[j]==0)
        return false;
    else
        return true;
}
```

基于 BFS 遍历的算法如下：

```
bool BFSTrave(AdjGraph *G,int i,int j)
{   int k;
    int visited[MAXV];
    for (k=0;k<G->n;k++)
        visited[k]=0;
    BFS(G,i);                                   //从顶点 i 开始进行广度优先遍历
    if (visited[j]==0)
        return false;
    else
        return true;
}
```

6.【图的存储结构算法】假设图采用邻接表 $G1$ 存储，设计一个算法由 $G1$ 产生该图的逆邻接表 $G2$。

解：先分配逆邻接表 $G2$ 的存储空间，置 $G2->n=G1->n$，$G2->e=G1->e$，并置所有 $G2->$adjlist[i].firstarc 为空。用 i 扫描 $G1$ 的所有单链表：对于 $G1->$adjlist[i]中的结点 p，其顶点编号为 j，建立一个结点 q，置 $q->$adjvex$=i$，$q->$weight$=p->$weight，将 q 结点插入到 $G2->$adjlist[j]边单链表的开头。对应的算法如下：

```
void ReAdj(AdjGraph *G1,AdjGraph *&G2)
{   int i,j;
    ArcNode *p,*q;
```

```
    G2 = (AdjGraph *)malloc(sizeof(AdjGraph));
    G2->n = G1->n; G2->e = G1->e;
    for (i = 0;i < G2->n;i++)
        G2->adjlist[i].firstarc = NULL;
    for (i = 0;i < G1->n;i++)
    {   p = G1->adjlist[i].firstarc;
        while (p!= NULL)
        {   j = p->adjvex;
            q = (ArcNode *)malloc(sizeof(ArcNode));
            q->adjvex = i;
            q->weight = p->weight;
            q->nextarc = G2->adjlist[j].firstarc;
            G2->adjlist[j].firstarc = q;
            p = p->nextarc;
        }
    }
}
```

7.【邻接矩阵+DFS算法】假设图采用邻接矩阵表示，设计一个从顶点 v 出发的深度优先遍历算法。

解：设置全局数组 visited 并将所有元素初始化为 0，访问顶点 v 并置 visited[v]=1。在邻接矩阵 g.edges 的第 v 行找一个相邻点 w，若它没有访问过，递归调用 MDFS(g,w)从顶点 w 开始进行深度优先遍历。对应的算法如下：

```
int visited[MAXV];                  //全局变量
void MDFS(MatGraph g, int v)
{   int w;
    printf("%d  ",v);               //访问顶点 v
    visited[v] = 1;                 //置访问标记
    for (w = 0;w < g.n;w++)         //找顶点 v 的所有邻接点
        if (g.edges[v][w]!= 0 && g.edges[v][w]!= INF && visited[w] == 0)
            MDFS(g,w);              //找顶点 v 的未访问过的邻接点 w
}
```

8.【邻接矩阵+BFS算法】假设图 G 采用邻接矩阵存储，给出图的从顶点 v 出发的广度优先遍历算法，并分析算法的时间复杂度。

解：利用一个环形队列 Qu，先访问根结点并将其进队。队不空时循环：出队一个顶点，将它所有相邻的没有访问过的顶点进队。本算法的时间复杂度为 $O(n^2)$。对应的算法如下：

```
void MBFS(MatGraph g, int v)
{   int Qu[MAXV], front = 0, rear = 0;          //定义循环队列并初始化
    int visited[MAXV];                          //定义存放结点的访问标志的数组
    int w, i;
    for (i = 0;i < g.n;i++) visited[i] = 0;     //访问标志数组初始化
    printf("%3d",v);                            //输出被访问顶点的编号
    visited[v] = 1;                             //置已访问标记
```

```
        rear = (rear + 1) % MAXV;
        Qu[rear] = v;                               //v 进队
        while (front!= rear)                        //若队列不空时循环
        {   front = (front + 1) % MAXV;
            w = Qu[front];                          //出队并赋给 w
            for (i = 0;i < g.n;i++)                 //找与顶点 w 相邻的顶点
                if (g.edges[w][i]!= 0 && g.edges[w][i]!= INF && visited[i] == 0)
                {                                   //若当前邻接顶点 i 未被访问
                    printf(" %3d",i);               //访问相邻顶点 i
                    visited[i] = 1;                 //置该顶点已被访问的标志
                    rear = (rear + 1) % MAXV;       //该顶点进队
                    Qu[rear] = i;
                }
        }
        printf("\n");
    }
```

9.【邻接矩阵+DFS 算法】假设图 G 采用邻接矩阵存储,设计一个算法采用深度优先遍历方法求有向图的根。若有向图中存在一个顶点 v,从 v 可以通过路径到达图中的其他所有顶点,则称 v 为该有向图的根。

解:由于从有向图的根出发可以到达图中的其他所有顶点,因此可以通过深度优先遍历方法来判断一个顶点是否为有向图的根。当采用深度优先遍历方法从顶点 i 出发能够访问所有顶点时表示 i 为图的根,若找到这样的顶点 i,返回 i,否则返回 -1。对应的算法如下:

```
    int visited[MAXV];                          //全局变量
    void MDFS(MatGraph g,int v)                 //基于邻接矩阵的深度优先遍历算法
    {   int w;
        visited[v] = 1;                         //置访问标记
        for (w = 0;w < g.n;w++)                 //找顶点 v 的所有邻接点
            if (g.edges[v][w]!= 0 && g.edges[v][w]!= INF && visited[w] == 0)
                MDFS(g,w);                      //找顶点 v 的未访问过的邻接点 w
    }
    int DGRoot(MatGraph g)                      //基于深度优先遍历求图的根
    {   int i,j,k,n;
        for (i = 0;i < g.n;i++)
        {   for (j = 0;j < g.n;j++)
                visited[j] = 0;
            MDFS(g,i);
            n = 0;                              //累计从顶点 i 出发访问到的顶点个数
            for (k = 0;k < g.n;k++)
                if (visited[k] == 1) n++;
            if (n == g.n) return(i);            //若访问所有顶点,则顶点 i 为根
        }
        return( - 1);                           //图没有根
    }
```

10.【邻接矩阵＋DFS 算法】假设图采用邻接矩阵存储。自由树（即无环连通图）$T=(V,E)$的直径是树中所有点对点间最短路径长度的最大值，即 T 的直径定义为 MAX $d(u,v)(u,v\in V)$，这里 $d(u,v)$表示顶点 u 到顶点 v 的最短路径长度（路径长度为路径中包含的边数）。设计一个算法求 T 的直径，以图 8.40 为例给出解，并分析算法的时间复杂度。

解：利用深度优先遍历求出一个根结点 v 到每个叶子结点的距离，这是由 Diameter(v)函数实现的，该函数的时间复杂度为 $O(n+e)$，n 为顶点个数，e 为边数。然后以每个顶点作为根结点调用 Diameter()函数，其中最大值即为 T 的直径，由此本算法的时间复杂度为$O(n(n+e))$。

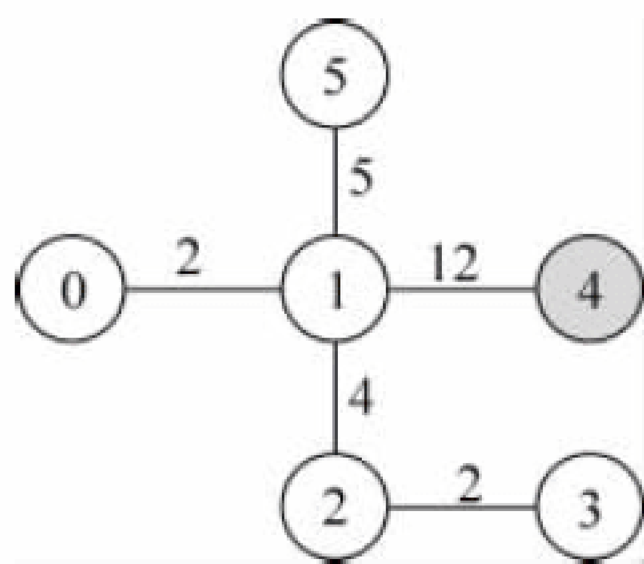


图 8.40　一棵自由树

设计 DFSTrav(g,v,w,len)算法通过形参 len 返回图 G 中从顶点 v 到以顶点 w 为根结点的子树中的所有叶子结点中的最大路径长度。例如，在图 8.40 中，顶点 0 到顶点 1 的返回值为 14（路径是 0-1-4），顶点 2 到顶点 1 的返回值为 16（路径是 2-1-4）。

设计 Diameter(g,v)算法返回图 G 中任意两个叶子结点经过 v 的最短路径长度的最大值，其方法是通过调用 DFSTrav(g,v,＊,len)找出两个最大的 len1 和 len2，它们都是图 G 中从顶点 v 到某个叶子结点的最大路径长度，且是不同的路径，则 len1＋len2 就是两个叶子结点经过顶点 v 的最短路径长度的最大值。例如，图 8.40 中经过顶点 1 的最短路径长度的最大值为 18，其路径为 3-2-1-4。

对应的算法如下：

```
void DFSTrav(MatGraph g,int parent,int child,int &len)
{   int clen,v = 0,maxlen;
    clen = len;
    maxlen = len;
    while (v < g.n && g.edges[child][v] == 0)   //找 child 的第一个邻接点 v
        v++;
    while (v < g.n)                             //存在邻接点时循环
    {   if (v!= parent)
        {   len = len + g.edges[child][v];
            DFSTrav(g,child,v,len);
            if (len > maxlen)                   //比较找最大值
                maxlen = len;
        }
        v++;
        while (v < g.n && g.edges[child][v] == 0)   //找 child 的下一个邻接点
            v++;
        len = clen;
    }
    len = maxlen;
}
int Diameter(MatGraph g,int v)
{   int maxlen1 = 0;                            //存放目前找到的根 v 到叶子结点的最大值
    int maxlen2 = 0;                            //存放目前找到的根 v 到叶子结点的次大值
    int len = 0;                                //记录深度优先遍历中到某个叶子结点的距离
```

```
    int w = 0;                                        //存放 v 的邻接顶点
    while (w < g.n && g.edges[v][w] == 0)             //找与 v 相邻的第一个顶点 w
        w++;
    while (w < g.n)                                   //存在邻接点时循环
    {   len = g.edges[v][w];
        DFSTrav(g,v,w,len);
        if (len > maxlen1)
        {   maxlen2 = maxlen1;
            maxlen1 = len;
        }
        else if (len > maxlen2)
            maxlen2 = len;
        w++;
        while (w < g.n && g.edges[v][w] == 0)
            w++;                                      //找 v 的下一个邻接点 w
    }
    return maxlen1 + maxlen2;
}
int MaxDiameter(MatGraph g)                           //求 g 的直径
{   int i,diam,d;
    diam = Diameter(g,0);
    for (i = 1;i < g.n;i++)                           //找出从所有顶点出发直径的最大值
    {   d = Diameter(g,i);
        if (diam < d) diam = d;
    }
    return diam;
}
```

设计以下主函数：

```
int main()
{   MatGraph g;
    int A[MAXV][MAXV] = {{0,2,0,0,0,0},{2,0,4,0,12,5},{0,4,0,2,0,0},
        {0,0,2,0,0,0},{0,12,0,0,0,0},{0,5,0,0,0,0}};
    int n = 6,e = 5;
    CreateMat(g,A,n,e);            //建立图 8.40 所示的邻接矩阵
    printf("图 G 的邻接矩阵:\n");DispMat(g);
    printf("T 的直径 = %d\n",MaxDiameter(g));
    return 1;
}
```

程序的执行结果如下：

```
图 G 的邻接矩阵:
   0   2   0   0   0   0
   2   0   4   0  12   5
   0   4   0   2   0   0
   0   0   2   0   0   0
   0  12   0   0   0   0
   0   5   0   0   0   0
T 的直径 = 18
```

11.【图的最短路径算法】给定 n 个村庄之间的交通图。若村庄 i 与村庄 j 之间有路可通，则将顶点 i 与顶点 j 之间用边连接，边上的权值 w_{ij} 表示这条道路的长度。现打算在这 n 个村庄中选定一个村庄建一所医院，设计一个算法求出该医院应建在哪个村庄才能使距离医院最远的村庄到医院的路程最短。

解：将 n 个村庄的交通图用二维数组 A 表示。算法思路是先应用 Floyd 算法计算每对顶点之间的最短路径；找出从每一个顶点到其他各顶点的最短路径中最长的路径；最后在这 n 条最长路径中找出最短的一条。对应的算法如下：

```
int MaxMinPath(MatGraph g)
{    int i,j,k;
     int A[MAXV][MAXV];
     int s,min = 32767;
     for (i = 0;i < g.n;i++)
          for (j = 0;j < g.n;j++)
               A[i][j] = g.edges[i][j];
     for (k = 0;k < g.n;k++)              //应用 Floyd 算法计算每对村庄之间的最短路径长度
          for (i = 0;i < g.n;i++)
               for (j = 0;j < g.n;j++)
                    if (A[i][k] + A[k][j]< A[i][j])
                         A[i][j] = A[i][k] + A[k][j];
     k = -1;
     for (i = 0;i < g.n;i++)              //对每个村庄循环一次
     {    s = 0;
          for (j = 0;j < g.n;j++)         //求到达村庄 i 的一条最长路径长度
               if (A[j][i]> s)
                    s = A[j][i];
          if (s < min)                    //在各最长路径中选最短的一条,将该村庄放在 k 中
          {    k = i;
               min = s;
          }
     }
     return k;
}
```

对于图 8.35，上述算法求出的结果是顶点 3。此题若改成求该医院应建在哪个村庄使其他所有村庄到医院的往返路径总和最短，则算法改为：

```
int MinPath(MatGraph g)
{    int i,j,k;
     int A[MAXV][MAXV];
     int min = 32767,B[MAXV];
     for (i = 0;i < g.n;i++)
          for (j = 0;j < g.n;j++)
               A[i][j] = g.edges[i][j];
     for (k = 0;k < g.n;k++)              //应用 Floyd 算法计算每对村庄之间的最短路径长度
          for (i = 0;i < g.n;i++)
               for (j = 0;j < g.n;j++)
                    if (A[i][k] + A[k][j]< A[i][j])
```

```
                    A[i][j] = A[i][k] + A[k][j];
    for (i = 0;i < g.n;i++)            //求每个村庄到村庄 i 的往返路径长度
    {   B[i] = 0;
        for (j = 0;j < g.n;j++)
            B[i] += A[i][j] + A[j][i];
    }
    for (i = 0;i < g.n;i++)            //求最短往返路径长度的顶点 k
        if (B[i]< min)
        {   k = i;
            min = B[i];
        }
    return k;
}
```

对于图 8.35,上述算法求出的结果是顶点 3。

12.【图的最短路径算法】假设一个带权有向图采用邻接表存储,设计求源点 v 到其他顶点最短路径和最短路径长度的 Dijkstra 算法。不考虑路径输出,说明算法的时间复杂度。

解:利用 Dijkstra 算法过程和邻接表的特点设计对应的算法如下。

```
void Dispath(AdjGraph *G,int dist[],int path[],int S[],int v)
//输出从顶点 v 出发的所有最短路径
{   int i,j,k;
    int apath[MAXV],d;                    //存放一条最短路径(逆向)及其顶点个数
    for (i = 0;i < G->n;i++)              //循环输出从顶点 v 到 i 的路径
        if (S[i] == 1 && i!= v)
        {   printf("  从顶点 %d 到顶点 %d 的路径长度为: %d\t 路径为:",v,i,dist[i]);
            d = 0; apath[d] = i;          //添加路径上的终点
            k = path[i];
            if (k == -1)                  //没有路径的情况
                printf("无路径\n");
            else                          //存在路径时输出该路径
            {   while (k!= v)
                {   d++; apath[d] = k;
                    k = path[k];
                }
                d++; apath[d] = v;        //添加路径上的起点
                printf("%d",apath[d]);    //先输出起点
                for (j = d-1;j >= 0;j--)  //再输出其他顶点
                    printf(",%d",apath[j]);
                printf("\n");
            }
        }
}
void MDijkstra(AdjGraph *G,int v)         //基于邻接表的 Dijkstra 算法
{   ArcNode *p;
    int dist[MAXV],path[MAXV];
    int S[MAXV];                      //S[i] = 1 表示顶点 i 在 S 中, S[i] = 0 表示顶点 i 在 U 中
```

```
    int Mindis,i,j,u;
    for (i=0;i<G->n;i++)              //距离初始化为∞,S置为空,path置为-1
    {   dist[i]=INF;
        S[i]=0;
        path[i]=-1;
    }
    S[v]=1;                           //将源点v放入S
    p=G->adjlist[v].firstarc;
    while (p!=NULL)                   //设置dist[p->adjvex]为<v,p->adjvex>的权值
    {   dist[p->adjvex]=p->weight;
        path[p->adjvex]=v;
        p=p->nextarc;
    }
    for (i=0;i<G->n-1;i++)            //循环直到所有顶点的最短路径都求出
    {   Mindis=INF;                   //Mindis置最大长度初值
        for (j=0;j<G->n;j++)          //选取不在S中且具有最短路径长度的顶点u
            if (S[j]==0 && dist[j]<Mindis)
            {   u=j;
                Mindis=dist[j];
            }
        S[u]=1;                       //顶点u加入S中
        p=G->adjlist[u].firstarc;
        while (p!=NULL)
        {   j=p->adjvex;            //顶点u的出边邻接点为j,该边的权值为p->weight
            if (S[j]==0 && dist[u]+p->weight<dist[j])
            {                         //修改不在S中的顶点的最短路径
                dist[j]=dist[u]+p->weight;
                path[j]=u;
            }
            p=p->nextarc;
        }
    }
    Dispath(G,dist,path,S,v);         //输出最短路径
}
```

设计以下主程序：

```
int main()
{   AdjGraph *G;
    int A[MAXV][MAXV]={
        {0,4,6,6,INF,INF,INF},
        {INF,0,1,INF,7,INF,INF},
        {INF,INF,0,INF,6,4,INF},
        {INF,INF,2,0,INF,5,INF},
        {INF,INF,INF,INF,0,INF,6},
        {INF,INF,INF,INF,1,0,8},
        {INF,INF,INF,INF,INF,INF,0}};
    int n=7, e=12;
    CreateAdj(G,A,n,e);             //建立《教程》中图8.35所示的邻接表
```

```
    int v = 0;
    printf("从%d顶点出发的最短路径如下:\n",v);
    MDijkstra(G,v);
    DestroyAdj(G);
    return 1;
}
```

程序的执行结果如下：

```
从0顶点出发的最短路径如下:
  从顶点0到顶点1的路径长度为:4         路径为:0,1
  从顶点0到顶点2的路径长度为:5         路径为:0,1,2
  从顶点0到顶点3的路径长度为:6         路径为:0,3
  从顶点0到顶点4的路径长度为:10        路径为:0,1,2,5,4
  从顶点0到顶点5的路径长度为:9         路径为:0,1,2,5
  从顶点0到顶点6的路径长度为:16        路径为:0,1,2,5,4,6
```

第 9 章 查找

9.1 本章知识体系

1. 知识结构图

本章的知识结构如图 9.1 所示。



图 9.1　第 9 章知识结构图

2. 基本知识点

（1）顺序查找算法及其性能分析。

（2）折半查找算法及其性能分析。

（3）索引存储结构和分块查找的特点及其性能分析。

（4）二叉排序树的查找和插入算法设计，删除过程。

（5）平衡二叉树的调整过程和性能分析。

（6）B－树的查找、插入和删除过程。

（7）哈希表的特点、哈希函数的构造方法和解决冲突的方法。

（8）各种查找方法的特点和性能对比分析。

3. 要点归纳

（1）在查找表中，所有记录的关键字是唯一的。查找表分为静态查找表和动态查找表。

（2）衡量查找算法性能的主要指标是平均查找长度（ASL），分为成功和不成功平均查找长度。

（3）在对线性表进行顺序查找时，线性表既可以采用顺序存储，也可以采用链式存储。

(4) 在对线性表进行折半查找时,要求线性表必须以顺序方式存储,且元素按关键字有序排列。

(5) 在对含有 n 个元素的有序顺序表进行顺序查找时,算法时间复杂度仍然为 $O(n)$。

(6) 在对含有 n 个元素的有序顺序表进行折半查找时,算法时间复杂度是 $O(\log_2 n)$。

(7) 折半查找的判定树反映了所有可能的查找情况,内部结点对应查找成功,外部结点对应查找失败。若内部结点个数为 n,则外部结点恰好有 $n+1$ 个。

(8) 在对含有 n 个元素的有序顺序表进行折半查找时,成功和不成功情况下关键字比较的最多次数为 $\lceil \log_2(n+1) \rceil$。

(9) 分块查找在等概率情况下,其平均查找长度不仅与表长有关,而且与每一块中的元素个数有关。分块查找算法的效率介于顺序查找和折半查找之间。

(10) 向一棵二叉排序树中插入一个结点所需的关键字比较次数最多是树的高度。

(11) 向一棵二叉排序树中插入一个结点均是以叶子结点插入的。

(12) 若先删除二叉排序树中的某个结点,再重新插入该结点,不一定得到原来的二叉排序树。

(13) 二叉排序树的中序序列是一个递增有序序列。

(14) 平衡二叉树的查找过程和二叉排序树的查找过程相同。插入过程是先采用二叉排序树的方法插入关键字 k,若不平衡则需要调整,调整分为 LL、RR、LR 和 RL 类型。

(15) 给定结点个数的平衡二叉树的高度不一定是唯一的。

(16) 哈希表不同于其他存储方法,它根据元素的关键字直接计算出该元素的存储地址。这个地址计算函数就是哈希函数。

(17) 设计哈希表主要是设计哈希函数和哈希冲突解决方法。

(18) 同义词是指两个不同关键字的元素,其哈希函数值相同,这种冲突称为同义词冲突。非同义词冲突是指哈希函数值不相同的两个元素争夺同一个后继哈希地址,导致出现堆积(或聚集)现象。

(19) 在采用线性探测法处理冲突的哈希表中,所有同义词在表中不一定相邻。

(20) 在理想的情况下(如元素个数和元素值确定,可以设计出不出现冲突的哈希函数),在哈希表中查找一个元素的时间复杂度为 $O(1)$。

9.2 教材中的练习题及参考答案

1. 设有 5 个数据 do、for、if、repeat、while,它们排在一个有序表中,其查找概率分别是 $p_1=0.2$,$p_2=0.15$,$p_3=0.1$,$p_4=0.03$,$p_5=0.01$,而查找它们之间不存在数据的概率分别为 $q_0=0.2$,$q_1=0.15$,$q_2=0.1$,$q_3=0.03$,$q_4=0.02$,$q_5=0.01$,该有序表如下:

do	for	if	repeat	while

q_0 p_1 q_1 p_2 q_2 p_3 q_3 p_4 q_4 p_5 q_5

(1) 试画出对该有序表分别采用顺序查找和折半查找时的判定树。

(2) 分别计算顺序查找的查找成功和不成功的平均查找长度。

(3) 分别计算折半查找的查找成功和不成功的平均查找长度。

答:(1) 对该有序表分别采用顺序查找和折半查找时的判定树分别如图 9.2 和 9.3 所示。

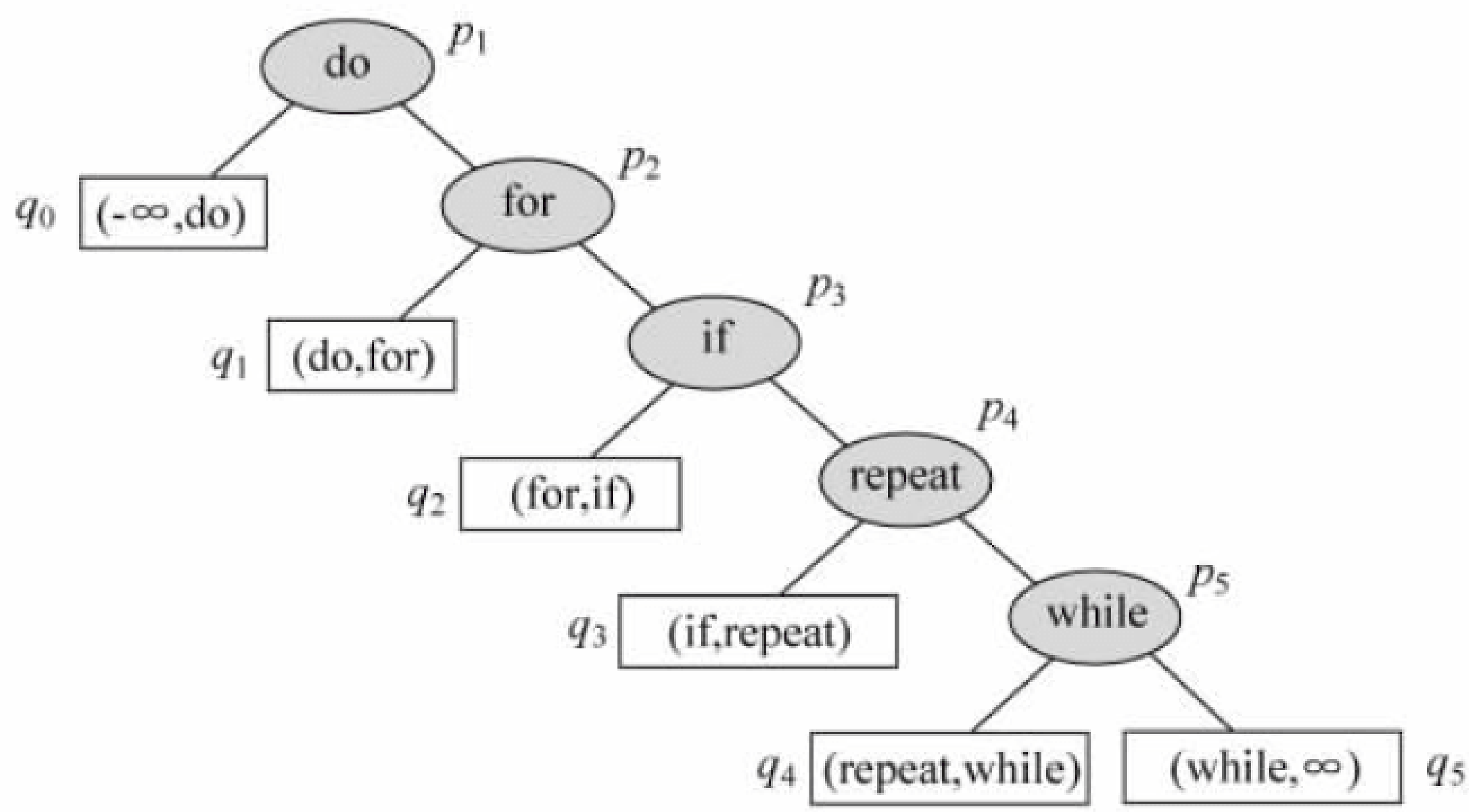


图 9.2　有序表上顺序查找的判定树

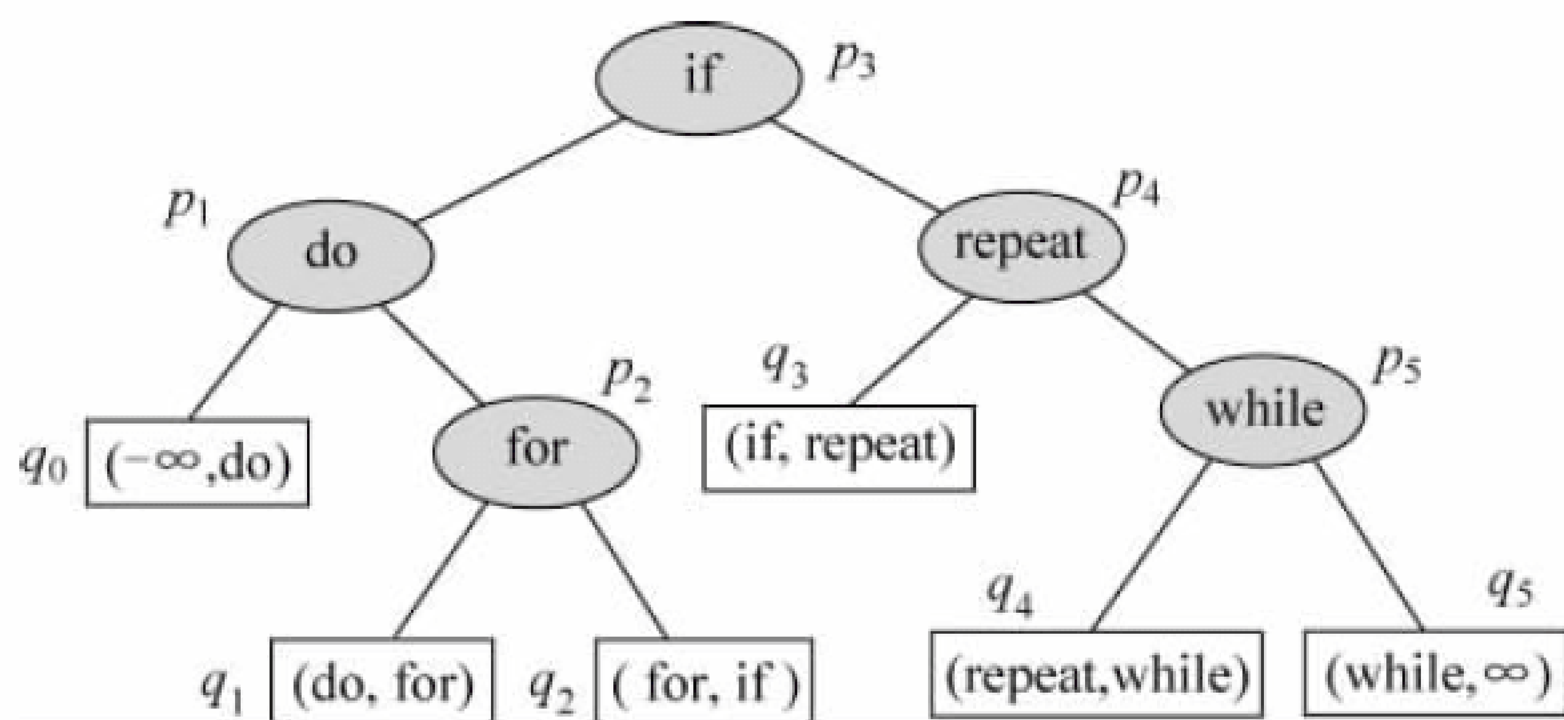


图 9.3　有序表上折半查找的判定树

(2) 对于顺序查找,成功查找到第 i 个元素需要 i 次比较,不成功查找需要比较的次数为对应外部结点的层次减 1:

$ASL_{成功}=(1p_1+2p_2+3p_3+4p_4+5p_5)=0.97$。

$ASL_{不成功}=(1q_0+2q_1+3q_2+4q_3+5q_4+5q_5)=1.07$。

(3) 对于折半查找,成功查找需要比较的次数为对应内部结点的层次,不成功查找需要比较的次数为对应外部结点的层次减 1:

$ASL_{成功}=(1p_3+2(p_1+p_4)+3(p_2+p_5))=1.04$。

$ASL_{不成功}=(2q_0+3q_1+3q_2+2q_3+3q_4+3q_5)=1.3$。

2. 对于 $A[0..10]$有序表,在等概率的情况下,求采用折半查找法时成功和不成功的平均查找长度。对于有序表(12,18,24,35,47,50,62,83,90,115,134),当用折半查找法查找 90 时需要进行多少次查找可确定成功?查找 47 时需要进行多少次查找可确定成功?查找 100 时需要进行多少次查找才能确定不成功?

答:对于 $A[0..10]$有序表构造的判定树如图 9.4(a)所示。因此有:

$$ASL_{成功}=\frac{1\times1+2\times2+4\times3+4\times4}{11}=3$$

$$ASL_{不成功} = \frac{4 \times 3 + 8 \times 4}{12} = 3.67$$

对于题中给定的有序表构造的判定树如图 9.4(b)所示。在查找 90 时，关键字比较次序是 50、90，比较两次。在查找 47 时，关键字比较次序是 50、24、35、47，比较 4 次。在查找 100 时，关键字比较次序是 50、90、115，比较 3 次。

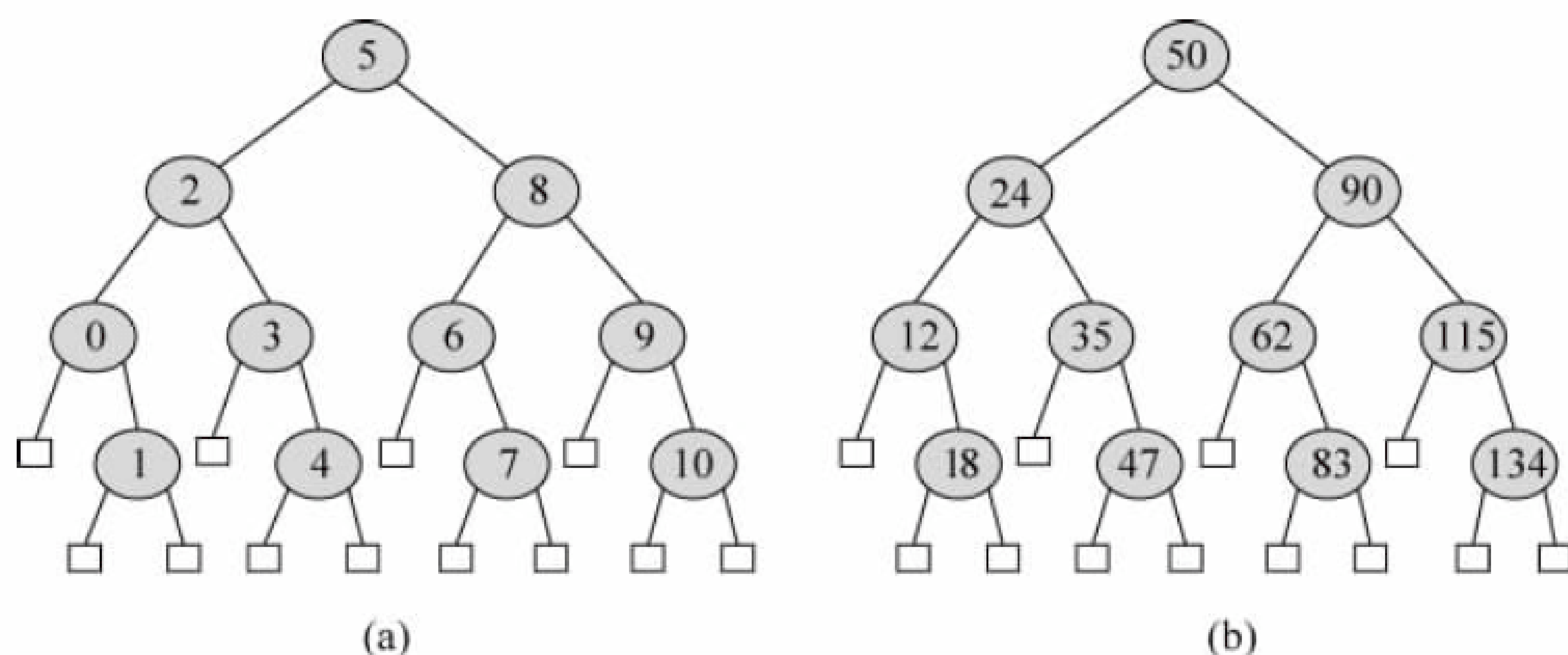


图 9.4　两棵判定树

3. 有以下查找算法：

```
int fun(int a[],int n,int k)
{   int i;
    for (i=0;i<n;i+=2)
        if (a[i]==k)
            return i;
    for (i=1;i<n;i+=2)
        if (a[i]==k)
            return i;
    return -1;
}
```

(1) 指出 fun(a,n,k)算法的功能。

(2) 当 a[]={2,6,3,8,1,7,4,9}时，执行 fun(a,n,1)后的返回结果是什么？一共进行了几次比较？

(3) 当 a[]={2,6,3,8,1,7,4,9}时，执行 fun(a,n,5)后的返回结果是什么？一共进行了几次比较？

答：(1) fun(a,n,k)算法的功能是在数组 a[0..n−1]中查找元素值为 k 的元素，若找到了返回 k 对应元素的下标，否则返回−1。算法先在奇数序号的元素中查找，如没有找到，再在偶数序号的元素中查找。

(2) 当 a[]={2,6,3,8,1,7,4,9}时，执行 fun(a,n,1)后的返回结果是 4，表示查找成功。一共进行了 3 次比较。

(3) 当 a[]={2,6,3,8,1,7,4,9}时，执行 fun(a,n,5)后的返回结果是−1，表示查找不成功。一共进行了 8 次比较。

4. 假设一棵二叉排序树的关键字为单个字母，其后序遍历序列为 $ACDBFIJHGE$，回答以下问题：

（1）画出该二叉排序树。

（2）求在等概率下的查找成功的平均查找长度。

（3）求在等概率下的查找不成功的平均查找长度。

答：（1）该二叉排序树的后序遍历序列为 ACDBFIJHGE，则中序遍历序列为 ABCDEFGHIJ，由后序序列和中序序列构造的二叉排序树如图 9.5 所示。

（2）$ASL_{成功}=(1\times1+2\times2+4\times3+2\times4+1\times5)/10=3$。

（3）$ASL_{不成功}=(6\times3+3\times4+2\times5)/11=3.64$。

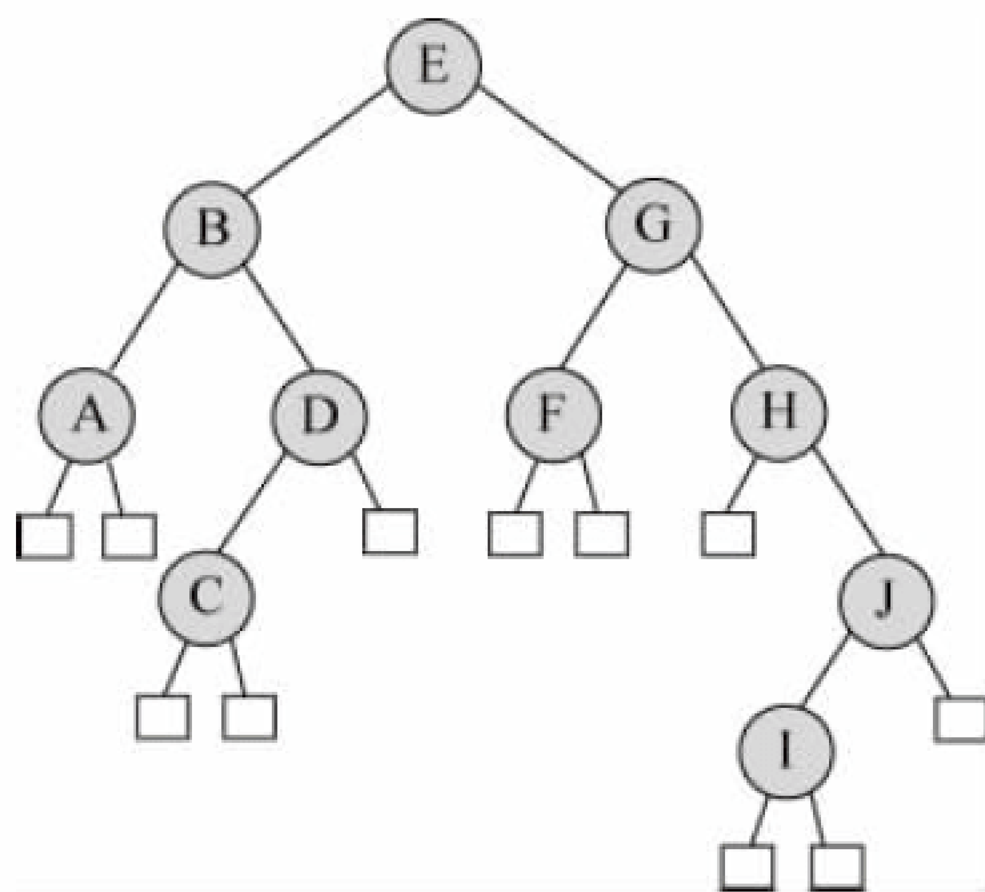


图 9.5　一棵二叉排序树

5. 证明如果一棵非空二叉树（所有结点值均不相同）的中序遍历序列是从小到大有序的，则该二叉树是一棵二叉排序树。

证明：对于关键字为 k 的任一结点 a，由中序遍历过程可知，在中序遍历序列中，它的左子树的所有结点的关键字排在 k 的左边，它的右子树的所有结点的关键字排在 k 的右边，由于中序序列是从小到大排列的，所以结点 a 的左子树中所有结点的关键字小于 k，结点 a 的右子树中所有结点的关键字大于 k，这满足二叉排序树的性质，所以该二叉树是一棵二叉排序树。

6. 由 23、12、45 关键字构成的二叉排序树有多少棵？其中属于平衡二叉树的有多少棵？

答：这里 $n=3$，构成的二叉排序树的个数 $=\dfrac{1}{n+1}C_{2n}^{n}=5$，如图 9.6 所示。

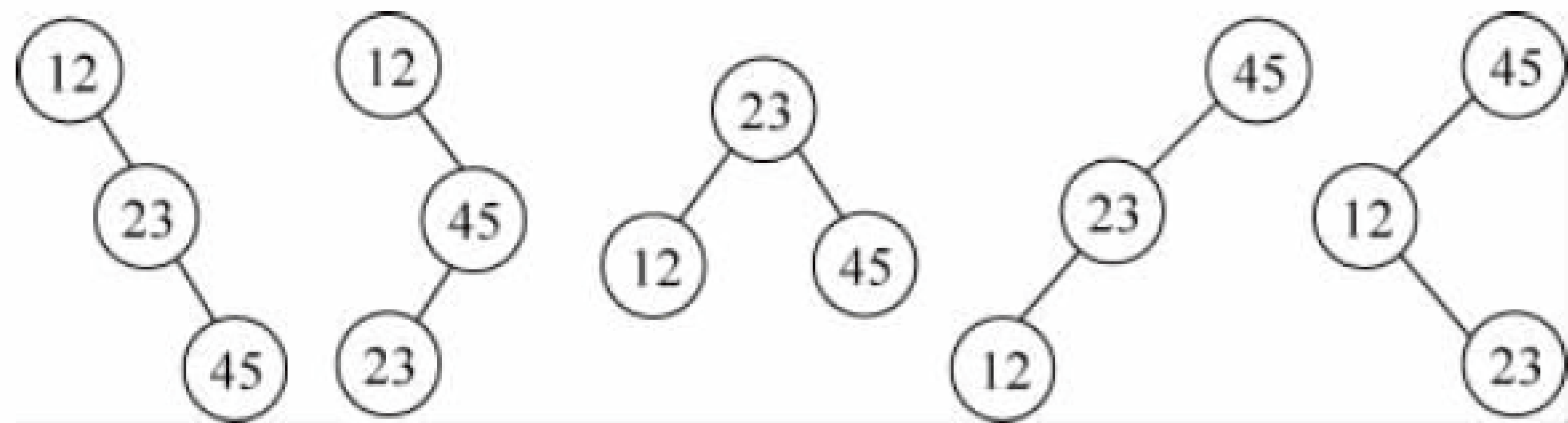


图 9.6　5 棵二叉排序树

其中的平衡二叉树有一棵，为图 9.6 中第 3 棵。

7. 将整数序列（4,5,7,2,1,3,6）中的元素依次插入到一棵空的二叉排序树中，试构造相应的二叉排序树，要求用图形给出构造过程。

答：构造一棵二叉排序树的过程如图 9.7 所示。

8. 将整数序列（4,5,7,2,1,3,6）中的元素依次插入到一棵空的平衡二叉树中，试构造相应的平衡二叉树，要求用图形给出构造过程。

答：构造一棵平衡二叉树的过程如图 9.8 所示。

9. 已知一棵 5 阶 B−树中有 53 个关键字，则树的最大高度是多少？

答：当每个结点的关键字个数都最少时，该 B−树的高度最大。根结点最少有一个关键字、两棵子树，第一层至少有一个结点。除根结点以外每个结点至少有 $\lceil 5/2\rceil-1=2$ 个关键字、3 棵子树，则第 2 层至少有两个结点，共 $2\times2=4$ 个关键字。第 3 层至少有 2×3 个

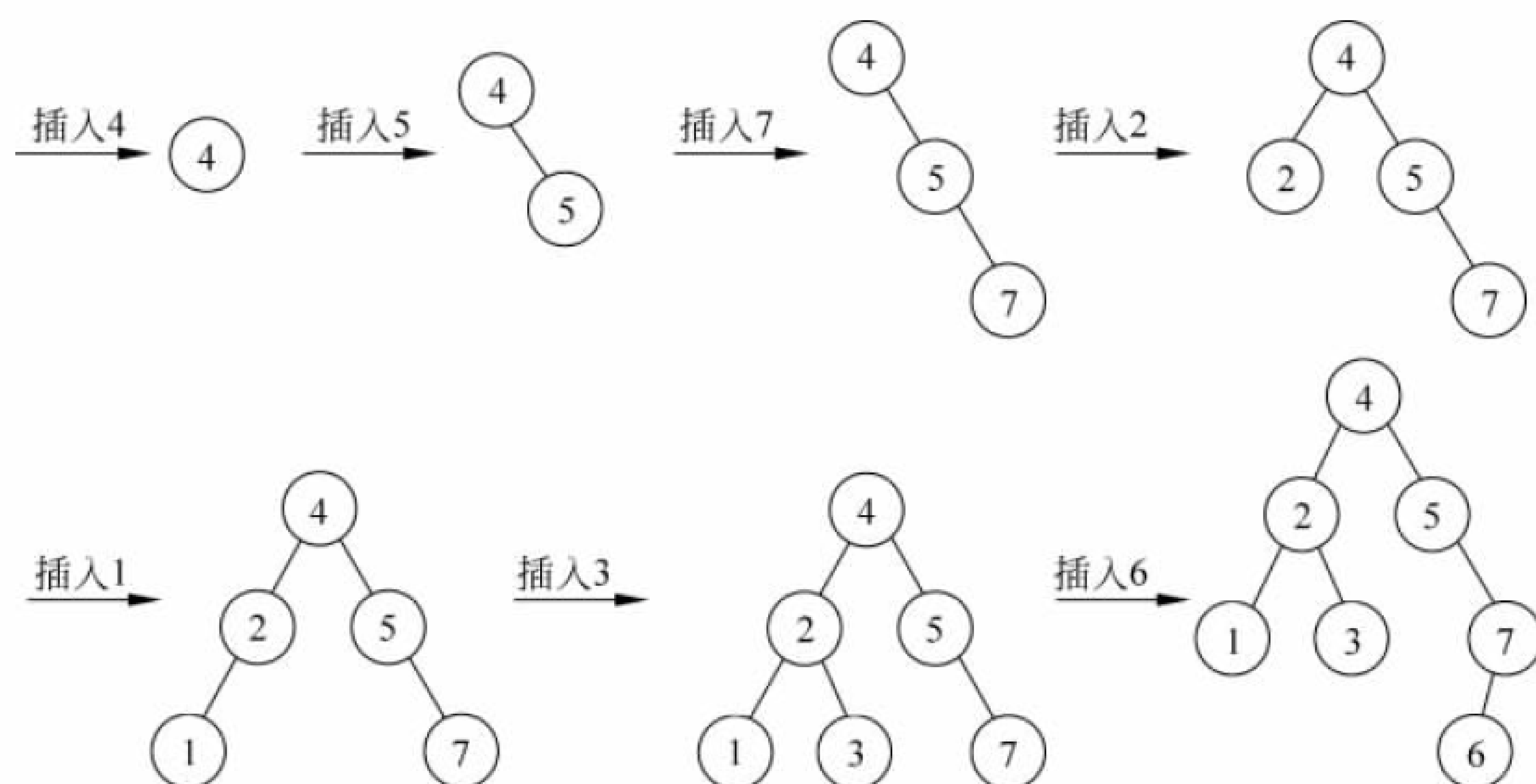


图 9.7 构造二叉排序树的过程

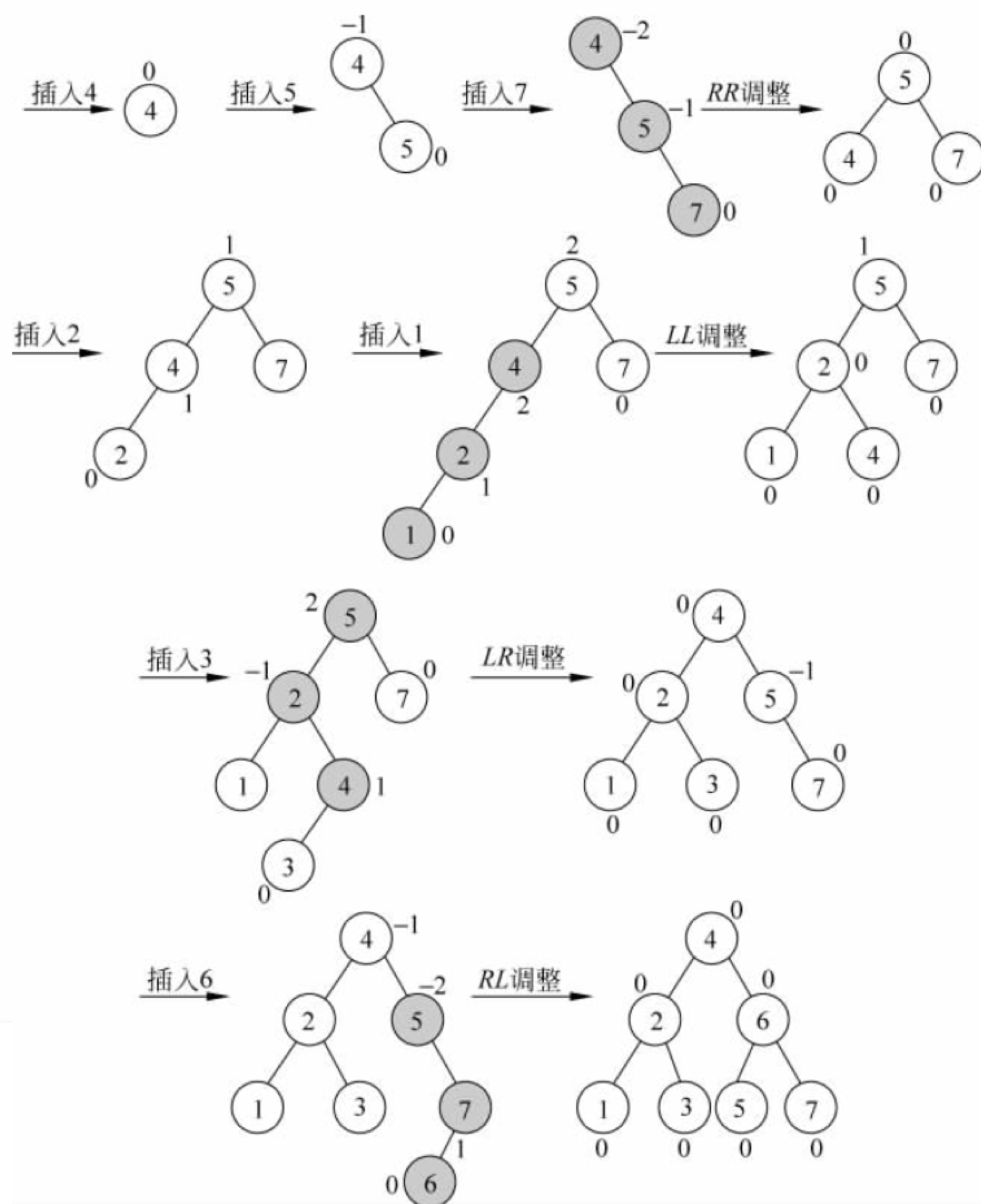


图 9.8 构造平衡二叉树的过程

结点，共 2×3×2=12 个关键字。第 4 层至少有 6×2 个结点，共 6×3×2=36 个关键字。而1+4+12+36=53，加上外部结点层，该 B－树的最大高度是 5 层。

10. 设有一组关键字(19,1,23,14,55,20,84,27,68,11,10,77)，其哈希函数为 $h(\text{key})=\text{key}\ \%\ 13$。采用开放地址法的线性探测法解决冲突，试在 0～18 的哈希表中对该关键字序列构造哈希表，并求在成功和不成功情况下的平均查找长度。

答：依题意，$m=19$，利用线性探测法计算下一地址的计算公式如下。

$$d_0 = h(\text{key})$$

$$d_{j+1} = (d_j + 1)\%m \quad j = 0,1,2,\cdots$$

计算各关键字存储地址的过程如下：

$h(19)=19\%13=6$，$h(1)=1\%13=1$，$h(23)=23\%13=10$

$h(14)=14\%13=1$(冲突)，$h(14)=(1+1)\%19=2$

$h(55)=55\%13=3$，$h(20)=20\%13=7$

$h(84)=84\%13=6$(冲突)，$h(84)=(6+1)\%19=7$(仍冲突)，$h(84)=(7+1)\%19=8$

$h(27)=27\%13=1$(冲突)，$h(27)=(1+1)\%19=2$(仍冲突)，$h(27)=(2+1)\%19=3$(仍冲突)，$h(27)=(3+1)\%19=4$

$h(68)=68\%13=3$(冲突)，$h(68)=(3+1)\%19=4$(仍冲突)，$h(68)=(4+1)\%19=5$

$h(11)=11\%13=11$

$h(10)=10\%13=10$(冲突)，$h(10)=(10+1)\%19=11$(仍冲突)，$h(10)=(11+1)\%19=12$

$h(77)=77\%13=12$(冲突)，$h(77)=(12+1)\%19=13$

因此，构建的哈希表如表 9.1 所示。

表 9.1　哈希表

下标	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
key		1	14	55	27	68	19	20	84		23	11	10	77					
探测次数		1	2	1	4	3	1	1	3		1	1	3	2					

表中的探测次数即为相应关键字成功查找时所需比较关键字的次数，因此：

$$\text{ASL}_{成功} = (1+2+1+4+3+1+1+3+1+1+3+2)/12 = 1.92$$

查找不成功表示在表中未找到指定关键字的记录。以哈希地址是 0 的关键字为例，由于此处关键字为空，只需比较一次便可确定本次查找不成功；以哈希地址是 1 的关键字为例，若该关键字不在哈希表中，需要将它与 1～9 地址的关键字相比较，由于地址 9 的关键字为空，所以不再向后比较，共比较 9 次，其他的以此类推，所以得到如表 9.2 所示的结果。

表 9.2　不成功查找的探测次数

下标	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
key		1	14	55	27	68	19	20	84		23	11	10	77					
探测次数	1	9	8	7	6	5	4	3	2	1	5	4	3	2	1	1	1	1	1

而哈希函数为 $h(\text{key})=\text{key}\%13$，所以只需考虑 $h(\text{key})=0$～12 的情况，即：

$$\begin{aligned}\text{ASL}_{不成功} &= (1+9+8+7+6+5+4+3+2+1+5+4+3)/13 \\ &= 58/13 = 4.46\end{aligned}$$

11. 设计一个折半查找算法，求查找到关键字为 k 的记录所需关键字的比较次数。假设 k 与 $R[i]$. key 比较得到 3 种情况，即 $k==R[i]$. key、$k<R[i]$. key 或者 $k>R[i]$. key，计为一次比较（在教材中讨论关键字比较次数时都是这样假设的）。

解：用 cnum 累计关键字的比较次数，最后返回其值。由于题目中的假设，实际上 cnum 是求在判定树中比较结束时的结点层次（首先与根结点比较，所以 cnum 初始化为 1）。对应的算法如下：

```
int BinSearch1(RecType R[],int n,KeyType k)
{   int low=0,high=n-1,mid;
    int cnum=1;                 //成功查找需要一次比较
    while (low<=high)
    {   mid=(low+high)/2;
        if (R[mid].key==k)
            return cnum;
        else if (k<R[mid].key)
            high=mid-1;
        else
            low=mid+1;
        cnum++;
    }
    cnum--;                     //不成功查找的比较次数需要减 1
    return cnum;
}
```

12. 设计一个算法，判断给定的二叉树是否为二叉排序树。假设二叉树中结点的关键字均为正整数且均不相同。

解：对于二叉排序树来说，其中序遍历序列为一个递增有序序列。因此，对给定的二叉树进行中序遍历，如果始终能保持前一个值比后一个值小，则说明该二叉树是一棵二叉排序树。对应的算法如下：

```
KeyType predt = -32768;         //predt 为全局变量,保存当前结点的中序前驱的值,初值为-∞
bool JudgeBST(BSTNode *bt)
{   bool b1,b2;
    if (bt==NULL)
        return true;
    else
    {   b1=JudgeBST(bt->lchild);    //判断左子树
        if (b1==false)              //左子树不是 BST,返回假
            return false;
        if (bt->key<predt)          //当前结点违反 BST 性质,返回假
            return false;
        predt=bt->key;
        b2=JudgeBST(bt->rchild);    //判断右子树
        return b2;
    }
}
```

13．设计一个算法，在一棵非空二叉排序树 bt 中求出指定关键字为 k 结点的层次。

解：采用循环语句边查找边累计层次 lv，当找到关键字为 k 的结点时返回 lv，否则返回 0。对应的算法如下：

```
int Level(BSTNode *bt,KeyType k)
{   int lv=1;                          //层次lv置初值1
    BSTNode *p=bt;
    while (p!=NULL && p->key!=k)       //二叉排序树未找完或未找到则循环
    {   if (k<p->key)
            p=p->lchild;               //在左子树中查找
        else
            p=p->rchild;               //在右子树中查找
        lv++;                          //层次增1
    }
    if (p!=NULL)                       //找到后返回其层次
        return lv;
    else
        return(0);                     //表示未找到
}
```

14．设计一个哈希表 ha[0..$m-1$]存放 n 个元素，哈希函数采用除留余数法 $H(\text{key})=\text{key}\ \%\ p(p\leqslant m)$，解决冲突的方法采用开放定址法中的平方探测法。

(1) 设计哈希表的类型。

(2) 设计在哈希表中查找指定关键字的算法。

解：哈希表为 ha[0..$m-1$]，存放 n 个元素，哈希函数为 $H(\text{key})=\text{key}\ \%\ p(p\leqslant m)$。平方探测法：$H_i=(H(\text{key})+d_i) \bmod m(1\leqslant i\leqslant m-1)$，其中 $d_i=1^2、-1^2、2^2、-2^2、\cdots$

(1) 设计哈希表的类型如下：

```
#define MaxSize 100                    //定义最大哈希表长度
#define NULLKEY -1                     //定义空关键字值
#define DELKEY -2                      //定义被删关键字值
typedef int KeyType;                   //关键字类型
typedef char * InfoType;               //其他数据类型
typedef struct
{   KeyType key;                       //关键字域
    InfoType data;                     //其他数据域
    int count;                         //探测次数域
} HashTable[MaxSize];                  //哈希表类型
```

(2) 对应的算法如下：

```
int SearchHT1(HashTable ha,int p,int m,KeyType k)      //在哈希表中查找关键字k
{   int adr,adr1,i=1,sign;
    adr=adr1=k % p;                                     //求哈希函数值
    sign=1;
    while (ha[adr].key!=NULLKEY && ha[adr].key!=k)      //找到的位置不空
    {   adr=(adr1+sign*i*i) % m;
```

```
        if (sign == 1)
            sign = - 1;
        else                        //sign == - 1
        {   sign = 1;
            i++;
        }
    }
    if (ha[adr].key == k)           //查找成功
        return adr;
    else                            //查找失败
        return - 1;
}
```

9.3 补充练习题及参考答案

9.3.1 单项选择题

1. 不适合在链式存储结构上实现的查找方法是________。

A. 顺序查找　　B. 折半查找

C. 二叉排序树查找　　D. 哈希查找

答：B。

2. 采用顺序查找方法查找长度为 n 的线性表时，不成功查找的平均查找长度为________。

A. n　　B. $n/2$　　C. $(n+1)/2$　　D. $(n-1)/2$

答：A。不成功的查找需要和表中每个元素比较一次。

3. 在数据元素有序、元素个数较多而且固定不变的情况下宜采用________法。

A. 折半查找　　B. 分块查找

C. 二叉排序树查找　　D. 顺序查找

答：A。

4. 有一个长度为 12 的有序表，按二分查找法对该表进行查找，在表内各元素等概率的情况下，查找成功所需的平均比较次数为________。

A. 35/12　　B. 37/12　　C. 39/12　　D. 43/12

答：B。

5. 有一个有序表 $R[1..13]=\{1,3,9,12,32,41,45,62,75,77,82,95,100\}$，当用二分查找法查找值为 82 的结点时，经过________次比较后查找成功。

A. 1　　B. 2　　C. 4　　D. 8

答：$n=13$，第 1 次与 $R[(1+13)/2=7]=45$ 比较，第 2 次与 $R[(8+13)/2=10]=77$，第 3 次与 $R[(11+13)/2=12]=95$ 比较，第 4 次与 $R[(10+12)/2=11]=85$ 比较，成功，总共比较 4 次，本题的答案为 C。

6. 设有 100 个元素的有序表,采用折半查找方法,成功时最多的比较次数是________。

A. 25　　B. 50　　C. 10　　D. 7

答:成功时最多的比较次数为$\lceil \log_2(n+1) \rceil = \lceil \log_2 101 \rceil = 7$。本题的答案为 D。

7. 设有 100 个元素的有序表,采用折半查找方法,不成功时最多的比较次数是________。

A. 25　　B. 50　　C. 10　　D. 7

答:不成功时最多的比较次数为$\lceil \log_2(n+1) \rceil = \lceil \log_2 101 \rceil = 7$。本题的答案为 D。

8. 在折半查找对应的判定树中,外部结点是________。

A. 一次成功查找过程终止的结点

B. 一次失败查找过程终止的结点

C. 一次成功查找过程中经过的中间结点

D. 一次失败查找过程中经过的中间结点

答:B。

9. 当采用分块查找时,数据的组织方式为________。

A. 数据分成若干块,每块内数据有序

B. 数据分成若干块,每块内数据无序,但块间必须有序,每块内最大(或最小)的数据组成索引块

C. 数据分成若干块,每块内数据有序,每块内最大(或最小)的数据组成索引块

D. 数据分成若干块,每块中的数据个数必须相同

答:B。

10. 在采用分块查找时,若线性表中共有 625 个元素,查找每个元素的概率相同,假设采用顺序查找来确定结点所在的块,则每块分为________个结点最佳。

A. 9　　B. 25　　C. 6　　D. 625

答:此时分块查找的最佳块数$=\sqrt{625}=25$。本题的答案为 B。

11. 设待查找元素为 47,且已存入变量 k 中,如果在查找过程中和 k 进行比较的元素依次是 47、32、46、25、47,则所采用的查找方法________。

A. 是一种错误的方法　　B. 可能是分块查找

C. 可能是顺序查找　　D. 可能是折半查找

答:如果是顺序查找或折半查找,第一次比较成功时就会结束。这里可能是分块查找,假设索引表是对块中最大的元素进行索引,先和索引表中的 47 比较找到相应块,然后到相应块(32、46、25、47)中查找。本题的答案为 B。

12. 如果在 n 个元素中查找其中任何一个元素至少要比较两次,则所用的查找方法有可能是________。

A. 折半查找　　B. 分块查找

C. 顺序查找　　D. 二叉排序树查找

答:分块查找方法先查找索引表,再查找数据表,至少要两次元素比较。本题的答案为 B。

13. 在二叉排序树中,凡是新插入的结点都是没有________的。

A. 孩子　　B. 关键字　　C. 平衡因子　　D. 赋值

答：在二叉排序树中，新结点都是作为叶子结点插入的。本题的答案为 A。

14. 以下关于二叉排序树的叙述中正确的是________。

A. 二叉排序树是动态树表，在插入新结点时会引起树的重新分裂和合并

B. 对二叉排序树进行层次遍历可以得到一个有序序列

C. 在构造二叉排序树时，若关键字序列有序，则二叉排序树的高度最大

D. 在二叉排序树中进行查找，关键字的比较次数不超过结点数的一半

答：C。

15. 关于二叉排序树的描述不正确的是________。

A. 二叉排序树的查找效率取决于树的形态

B. 从二叉排序树中删去一个结点后再重新插入，一定是作为叶子结点插入的

C. 在最坏情况下，利用插入操作构造一棵二叉排序树花费的代价为 $O(n\log_2 n)$

D. 在含有 n 个结点的平衡二叉排序树中，查找失败时最多花费代价为 $O(\log_2 n)$

答：在最坏情况下，利用插入操作构造一棵二叉排序树花费的代价为 $O(n^2)$，如构造一棵右单支二叉排序树就是这种情况。本题的答案为 C。

16. 图 9.9(a)所示的一棵二叉排序树不成功查找的平均查找长度是________。

A. 21/7　　B. 28/7　　C. 15/6　　D. 21/6

答：图 9.9(b)所示为带有外部结点的二叉排序树，其不成功查找的平均查找长度为 (2×2+3×3+4×2)/7=21/7。本题的答案为 A。

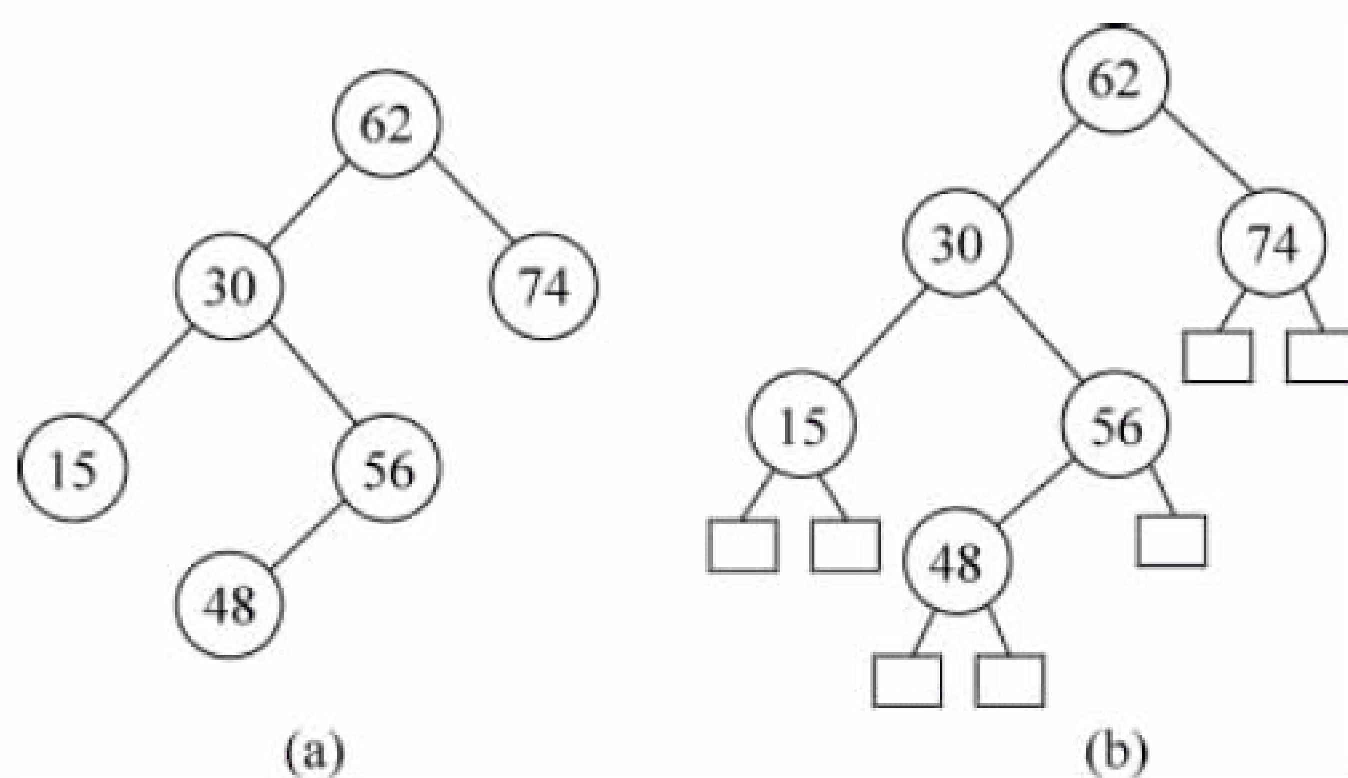


图 9.9　一棵二叉排序树

17. 在含有 27 个结点的二叉排序树上查找关键字为 35 的结点，则依次比较的关键字序列有可能是________。

A. 28,36,18,46,35　　B. 18,36,28,46,35

C. 46,28,18,36,35　　D. 46,36,18,28,35

答：考查每个关键字比较序列，看对应的查找树是否构成一棵二叉排序树的一部分。本题的答案为 D。

18. 对于下列关键字序列，不可能构成某二叉排序树中一条查找路径的序列是________。

A. 95,22,91,24,94,71　　B. 92,20,91,34,88,35

C. 21,89,77,29,36,38　　D. 12,25,71,68,33,34

答：考查每个关键字比较序列，看对应的查找树是否构成一棵二叉排序树的一部分。

本题的答案为 A。

19. 如图 9.10(a)所示的平衡二叉树插入关键字 48 后得到一棵新平衡二叉树，在新平衡二叉树中，关键字 37 所在结点的左、右子结点中保存的关键字分别是________。

A. 13,48　　B. 24,48　　C. 24,53　　D. 24,90

答：在平衡二叉树中插入关键字 48 后进行 RL 调整，如图 9.10(b)所示。本题的答案为 C。

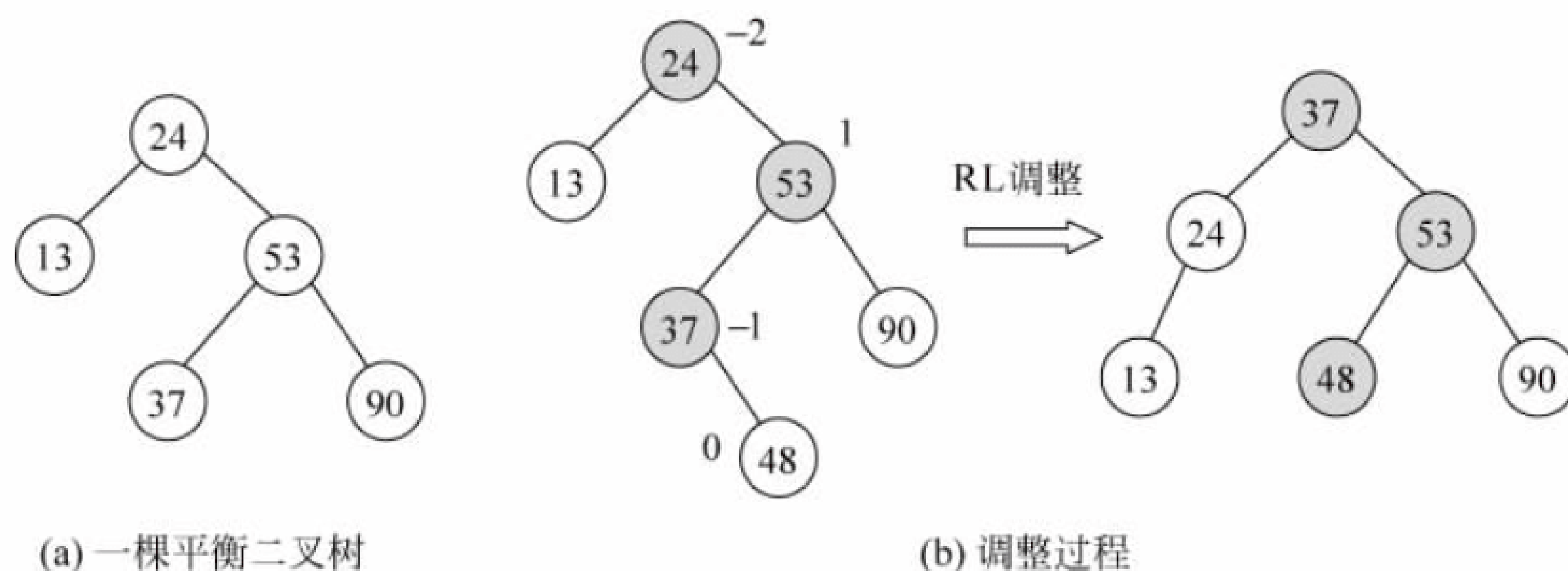


图 9.10　一棵平衡二叉树的插入和调整过程

20. 具有 5 层结点的 AVL 树至少有________个结点。

A. 10　　B. 12　　C. 15　　D. 17

答：设 N_h 表示高度为 h 的含有最少结点数的平衡二叉树，有以下关系。

$$N_0=0,\quad N_1=1,\quad N_2=2,\quad N_h=N_{h-1}+N_{h-2}+1$$

由此求出 $N_5=12$。本题的答案为 B。

21. 分别以下列序列构造平衡二叉树，与其他 3 个序列所构造结果不同的是________。

A. (4,2,3,1,6,5,7)　　B. (4,6,5,7,2,1,3)

C. (4,1,2,3,6,5,7)　　D. (4,2,1,3,6,5,7)

答：A。构造出各序列对应的平衡二叉树，然后进行比较。

22. 在含有 12 个结点的平衡二叉树上查找关键字为 35(存在该结点)的结点，则依次比较的关键字有可能是________。

A. 46,36,18,20,28,35　　B. 47,37,18,27,36

C. 27,48,39,43,37　　D. 15,45,25,35

答：设 N_h 表示高度为 h 的平衡二叉树中含有的最少结点数，有以下关系。

$$N_1=1,\quad N_2=2,\quad N_h=N_{h-1}+N_{h-2}+1$$

求出 $N_3=4$，$N_4=7$，$N_5=12$，也就是说，12 个结点的平衡二叉树中最大叶子结点的层数为 5，由于存在关键字为 35 的结点，所以最多比较 5 次一定能找到该结点，因此选项 A、B、C 错误(选项 B、C 表示没有查找到 35)。本题的答案为 D。

23. 在含有 15 个结点的平衡二叉树上查找关键字为 28 的结点，则依次比较的关键字有可能是________。

A. 30,36　　B. 38,48,28

C. 48,18,38,28　　D. 60,30,50,40,38,36

答：题中没有指出 AVL 中一定存在关键字为 28 的结点。设 N_h 表示高度为 h 的 AVL

中含有的最少结点数，有以下关系：

$$N_1=1,\quad N_2=2,\quad N_h=N_{h-1}+N_{h-2}+1\quad (当\ h>2)$$

求出 $N_6=20>15$，也就是说，高度为 6 的 AVL 最少有 20 个结点，因此 15 个结点的 AVL 的最大高度为 5。

另外，设 L_h 是高度为 h 的 AVL 中叶子结点的最小层次，有以下关系：

$$L_1=1,\quad L_2=2,\quad L_h=\mathrm{MIN}\{L_{h-1},L_{h-2}\}+1\quad (当\ h>2)$$

求出 $L_5=3$，也就是说，15 个结点的 AVL 中叶子结点的最小层次为 3，或者说外部结点的最小层次为 4。

选项 A 只有 2 次比较就确定查找失败，这是不可能的，因为至少需要 3 次比较才能落入某个外部结点中。选项 B 的查找过程不能构成二叉排序树的一部分，是错误的。选项 D 是一次失败查找，比较超过 5 次，因而错误。本题答案为 C。

24. 以下关于 m 阶 B－树的叙述中正确的是________。

A. 每个结点至少有两棵非空子树

B. 树中每个结点最多有 $\lceil m/2\rceil-1$ 个关键字

C. 所有外部结点均在同一层上

D. 当插入一个关键字引起 B－树结点分裂时树增高一层

答：C。

25. 已知一棵 3 阶 B－树中有 2047 个关键字，则树的最大高度是________。

A. 11　　B. 12　　C. 13　　D. 14

答：3 阶 B－树中每个内部结点至少包含 $\lceil 3/2\rceil-1=1$ 个关键字。当所有内部结点仅包含一个关键字时，B－树的高度最高，这样成为一棵满二叉树，高深度为 $\log_2(n+1)=\log_2(2047+1)=11$，加上外部结点层，共 12 层。本题的答案为 B。

26. 在一棵高度为 2（不计外部结点层）的 5 阶 B－树中所含关键字的个数最少是________。

A. 5　　B. 7　　C. 8　　D. 14

答：这里 $m=5$，内部结点共两层，根结点至少有两棵子树、一个关键字，两个孩子的关键字最少个数 $=\lceil m/2\rceil-1=2$，所以总计 5 个关键字。本题的答案为 A。

27. m 阶 B+树中除根结点以外，其他结点的关键字个数至少为________。

A. $\lceil m/2\rceil$　　B. $\lceil m/2\rceil-1$　　C. $\lceil m/2\rceil+1$　　D. 任意

答：A。

28. 下面关于 B－树和 B+树的叙述中不正确的结论是________。

A. B－树和 B+树都能有效地支持顺序查找

B. B－树和 B+树都能有效地支持随机查找

C. B－树和 B+树都是平衡的多分树

D. B－树和 B+树都可用于文件索引结构

答：由于 B+树的所有叶子结点中包含了全部关键字的信息，且叶子结点本身依关键字的大小自小而大顺序链接，可以进行顺序查找，而 B－树不支持顺序查找。本题的答案为 A。

29. 哈希表中出现哈希冲突是指________。

A. 两个元素具有相同的序号

B. 两个元素的关键字不同，而其他属性相同

C. 数据元素过多

D. 两个元素的关键字不同，而对应的哈希函数值相同

答：D。

30. 设哈希表长 $m=14$，哈希函数 $h(\text{key})=\text{key mod } 11$。表中已有 4 个元素，addr(15)=4，addr(38)=5，addr(61)=6，addr(84)=7，其余地址为空，如用二次探测法处理冲突，则关键字为 49 的结点的地址是________。

A. 8　　B. 3　　C. 5　　D. 9

答：addr(49)=49%11=5　冲突

$h_1=(5+1^2)\%11=6$　仍冲突

$h_2=(5-1^2)\%11=4$　仍冲突

$h_3=(5+2^2)\%11=9$

本题的答案为 D。

31. 下面有关哈希表的叙述中正确的是________。

A. 哈希查找的时间与规模 n 成正比

B. 不管是开放地址法还是拉链法，查找时间都与装填因子 α 有关

C. 开放地址法存在堆积现象，而拉链法不存在堆积现象

D. 拉链法中装填因子 α 必须小于 1

答：开放地址法中只有线性探测法容易产生堆积现象。本题的答案为 B。

32. 为提高哈希表的查找效率，可以采取的正确措施是________。

Ⅰ. 增大装填因子

Ⅱ. 设计冲突少的哈希函数

Ⅲ. 处理冲突时避免产生堆积现象

A. 仅Ⅰ　　B. 仅Ⅱ　　C. 仅Ⅰ、Ⅱ　　D. 仅Ⅱ、Ⅲ

答：装填因子 α 越大，发生冲突的可能性越大，所以Ⅰ错误。哈希表是由哈希函数和处理冲突两部分组成的，查找效率与这两部分有关，所以Ⅱ和Ⅲ是正确的。本题的答案为 D。

33. 在采用开放定址法解决冲突的哈希表中，发生堆积的原因主要是________。

A. 数据元素过多　　B. 装填因子 α 过大

C. 哈希函数选择不当　　D. 解决冲突的算法选择不当

答：堆积现象是指多个非同义词的元素争夺同一后继哈希地址，线性探测法容易产生堆积现象，而平方探测法不易产生堆积现象，所以说发生堆积的原因主要是解决冲突的算法选择不当造成的。本题的答案为 D。

34. 从 100 个元素中查找其中某个元素，如果最多进行 5 次元素之间的比较，则采用的查找方法只可能是________。

A. 折半查找　　B. 分块查找

C. 哈希查找　　D. 二叉排序树查找

答：$n=100$，折半查找的元素最多比较次数$=\lceil\log_2(n+1)\rceil=7$，分块查找和二叉排序树查找所需元素比较次数会更多，所以只有可能是哈希查找。本题的答案为 C。

35. 在顺序查找、折半查找、分块查找和二叉排序树中，在最坏情况下时间复杂度相同的是________。

A. 折半查找和二叉排序树查找　　B. 顺序查找和二叉排序树查找

C. 分块查找和二叉排序树查找　　D. 折半查找和分块查找

答：就最坏查找性能而言，顺序查找和二叉排序树查找的最坏时间复杂度均为 $O(n)$。本题的答案为 B。

9.3.2 填空题

1. 衡量查找算法性能好坏的主要标准是________。

答：关键字的平均比较次数或平均查找长度。

2. 采用顺序查找方法查找含 n 个元素的顺序表，若查找成功，则比较关键字的次数最多为___①___次；若查找不成功，则比较关键字的次数为___②___次。

答：① n　② n。

3. 在 n 个记录的有序顺序表中进行折半查找，查找过程落在对应判定树的第 i 层的某个外部结点中，则关键字的比较次数是________。

答：$i-1$。

4. 设有序表为{2,4,6,8,10,12,14,16,18,20}，采用折半查找方法查找元素 14，依次比较的元素是________。

答：10、16、12、14。

5. 设有序顺序表中有 $2^{20}-1$ 个记录，在采用折半查找时，不成功查找时的平均查找长度是________。

答：20。当 n 很大时，可以将对应的判定树看成是一棵满二叉树，所有的外部结点都在最底一层，它们的层次 $=\log_2(n+1)+1=21$，对应的记录比较次数等于外部结点的层次减 1，即 20。

6. 为了实现分块查找，线性表必须采用________方法存储。

答：索引。分块查找的数据组织由主数据表和索引表构成，属于一种索引存储结构。

7. 在分块查找方法中，首先查找___①___，然后查找相应的___②___。

答：① 索引表　② 块表。

8. 在某分块查找中，查找表中有 3100 个元素，共分为 31 块，则对应的索引表中的项数是___①___；如果索引表采用折半查找，则成功查找的平均查找长度为___②___。

答：①31　②54.5。每块对应一个索引项，共有 31 个索引项，每块有 3100/31=100 个元素。当索引表采用折半查找时，成功查找的平均查找长度 $=\log_2(31+1)-1=\log_2 32-1=4$，在每块中顺序查找，成功查找的平均查找长度 $=(100+1)/2=50.5$，所以分块查找的成功查找平均查找长度 $=4+50.5=54.5$。

9. 在高度为 h、含 n 个结点的二叉排序树上查找一个关键字的最多比较次数为________。

答：h。

10. 如果按关键字递增顺序依次将 n 个关键字插入到一棵初始为空的二叉排序树中，则对这样的二叉排序树查找时关键字的平均比较次数是________。

答：$(n+1)/2$。这样的二叉排序树是一个右单支树，此时查找退化为顺序查找，关键字的平均比较次数是$(n+1)/2$。

11. 对二叉排序树进行________遍历，可以得到按关键字从小到大排列的结点序列。

答：中序。

12. 对于两棵具有相同关键字集合而形状不同的二叉排序树，进行________遍历时可以得到完全相同的序列。

答：中序。

13. 对一棵二叉排序树进行这样的遍历：遍历右子树、访问根结点、遍历左子树，则得到的遍历序列是________。

答：按关键字递减排列的有序序列。

14. 在含有 n 个结点的二叉排序树中添加外部结点，则外部结点的个数为________。

答：$n+1$。

15. 输入序列为(20,35,30,…)，构造一棵平衡二叉树，其中的第一次调整为________型调整。

答：RL。

16. 输入序列为(20,11,12,…)，构造一棵平衡二叉树，其中的第一次调整为________型调整。

答：LR。

17. 在一个 10 阶的 B－树上，每个非根结点、非外部结点的结点中所含的关键字的数目最多允许为__①__个，最少允许为__②__个。

答：①9　②4。$m=10$，设非根结点、非外部结点的关键字个数为 j，则$\lceil m/2 \rceil-1 \leqslant j \leqslant m-1$，即 $4 \leqslant j \leqslant 9$。

18. 当向 B－树中插入一个关键字时可能引起结点的__①__，最终可能导致整个 B－树的高度__②__，当从 B－树中一个删除关键字时可能引起结点的__③__，最终可能导致整个 B－树的高度__④__。

答：①分裂　②增加 1　③合并　④减少 1。

19. 在采用哈希存储方法时，用于计算元素存储地址的是________。

答：哈希函数。

20. 评价哈希函数好坏的标准是________。

答：哈希函数的取值是否均匀。

21. 在各种查找方法中，其平均查找长度与结点个数 n 无关的查找方法是________。

答：哈希表查找法。哈希表查找的平均查找长度是装填因子 α 的函数。

22. 在哈希存储中，装填因子 α 的值越大，则__①__；α 的值越小，则__②__。

答：① 存取元素时发生冲突的可能性就越大　② 存取元素时发生冲突的可能性就越小。

9.3.3 判断题

1. 判断以下叙述的正确性。

(1) 顺序查找方法只能在顺序存储结构上进行。

（2）二分查找适合在有序的双链表上进行。

（3）分块查找的效率与查找表被分成多少块有关。

（4）二叉排序树是用来进行排序的。

（5）在二叉排序树中，每个结点的关键字都比左孩子关键字大，比右孩子关键字小。

（6）每个结点的关键字都比左孩子关键字大，比右孩子关键字小，这样的二叉树一定是二叉排序树。

（7）在二叉排序树中，新插入的关键字总是处于最底层。

（8）在二叉排序树中，新结点总是作为树叶来插入的。

（9）二叉排序树的查找效率和二叉排序树的高度有关。

（10）在平衡二叉排序树中，每个结点的平衡因子值都是相等的。

（11）在平衡二叉排序树中，以每个分支结点为根的子树都是平衡的。

（12）哈希存储方法只能存储数据元素的值，不能存储数据元素之间的关系。

（13）哈希冲突是指同一个关键字的记录对应多个不同的哈希地址。

（14）在哈希查找过程中，关键字的比较次数和哈希表中关键字的个数直接相关。

（15）在用线性探测法处理冲突的哈希表中，哈希函数值相同的关键字总是存放在一片连续的存储单元中。

答：（1）错误。顺序查找方法也可以在链式存储结构上进行。

（2）错误。二分查找适合具有随机查找的存储结构上进行，即适合在顺序存储的有序表上进行。

（3）正确。

（4）错误。二叉排序树主要用于改进一般二叉树的查找效率。

（5）正确。

（6）错误。对于二叉排序树，左子树上所有记录的关键字均小于根记录的关键字；右子树上所有记录的关键字均大于根记录的关键字，而不是仅仅与左、右孩子的关键字进行比较。

（7）错误。

（8）正确。

（9）正确。

（10）错误。每个结点的平衡因子的绝对值小于 2。

（11）正确。

（12）正确。每个元素的存储位置通过哈希函数和解决冲突的方法得到。

（13）错误。哈希冲突是指多个不同关键字的记录对应相同的哈希函数值。

（14）错误。只与装填因子直接相关。

（15）错误。不一定。

2. 判断以下叙述的正确性。

（1）用顺序表和单链表存储的有序表均适合采用二分查找方法来提高查找速度。

（2）n 个数据元素存放在一维数组 $R[1..n]$ 中，采用任何顺序查找方法，无论是有序还是无序，无论查找成功与否，算法的时间性能都是一样的。

（3）在二叉排序树的任意一棵子树中，关键字最小的结点必无左孩子，关键字最大的结

点必无右孩子。

(4) 二叉排序树的任意一棵子树也是二叉排序树。

(5) 在二叉排序树上删除一个结点时不必移动其他结点，只要将该结点的双亲结点的相应指针域置空即可。

(6) 对二叉排序树的查找都是从根结点开始的，则查找失败一定落在叶子上。

(7) 哈希表的查找效率主要取决于构造哈希表时选取的哈希函数和处理冲突的方法。

(8) 若哈希表的装填因子 $\alpha<1$，则可避免冲突的产生。

(9) 在哈希表中进行查找不需要关键字的比较。

(10) 在用线性探测法处理冲突的哈希表中，假设有 10 个记录互为同义词，把它们存入到哈希表中，总共最多需要进行 10 次探测。

答：(1) 错误。链表存储的有序表不适合采用二分查找。

(2) 错误。在顺序查找时可以利用有序性提高查找不成功的时间性能。

(3) 正确。

(4) 正确。

(5) 错误。如果删除的是非叶子结点，则必须调整某些结点，使调整后的二叉树仍为二叉排序树。

(6) 错误。查找失败一定落在外部结点上。

(7) 正确。

(8) 错误。α 越小只能说明发生冲突的概率越小，但仍有可能发生冲突。

(9) 错误。

(10) 错误。总共最少需要进行 $1+2+\cdots+10=55$ 次探测。

9.3.4 简答题

1. 试述顺序查找法、二分查找法和分块查找法对被查找表中元素的要求。对长度为 n 的表来说，3 种查找法在查找成功时的平均查找长度各是多少？

答：这 3 种方法对查找表的要求分别如下。

(1) 顺序查找法：表中元素可以任意次序存放。

(2) 二分查找法：表中元素必须以关键字的大小递增或递减的次序存放且以顺序表存储。

(3) 分块查找法：表中每块内的元素可任意次序存放，但块与块之间必须以关键字的大小递增(或递减)存放，即前一块内所有元素的关键字都不能大(或小)于后一块内任何元素的关键字。

这 3 种方法在查找成功时的平均查找长度分别如下。

(1) 顺序查找法：查找成功的平均查找长度为 $(n+1)/2$。

(2) 二分查找法：查找成功的平均查找长度为 $\log_2(n+1)-1$。

(3) 分块查找法：若用顺序查找确定所在的块，平均查找长度为 $(n/s+s)/2+1$；若用二分查找确定所在的块，平均查找长度为 $\log_2(n/s+1)+s/2$。其中，s 为每块含有的元素个数。

2. 为什么折半查找要求数据采用具有随机存取特性的存储结构来存储？

答：因为折半查找是按区间查找的，先将整个数据序列看成是一个区间，然后通过中位数比较缩小查找区间。这样在算法设计时要求通过上、下界快速确定所查找的区间，而只有具备随机存取特性的存储结构才能通过上、下标确定数据范围，而像链表等存储结构没有这个特性。

3. 对长度为 2^h-1 的有序表进行折半查找时，成功情况下最多比较多少次？查找失败的平均次数是多少？

答：对于长度为 2^h-1 的有序表，在进行折半查找时，对应判定树可以近似看成一棵高度为 h 的满二叉树（不含外部结点）。

成功情况下最多比较次数为判定树的高度，即 $\log_2(n+1)=\log_2(2^h-1+1)=h$。

当在这样的判定树中比较到外部结点时表示失败，所有外部结点对应的比较次数恰好等于该树的高度，即 $\log_2(n+1)=\log_2(2^h-1+1)=h$。

4. 有以下查找算法：

```
int fun(int a[ ],int low,int high,int k)
{   int mid;
    if (low <= high)
    {   mid = (low + high)/2;
        if (a[mid]<k)
            return fun(a,mid + 1,high,k);
        else if (a[mid]> k)
            return fun(a,low,mid - 1,k);
        else
            return mid;
    }
    else return( - 1);
}
```

（1）指出 fun(a,low,high,k)算法的功能。

（2）当 a[]={1,2,3,4,5,6,7,8}时，执行 fun(a,0,7,5)后的返回结果是什么？

答：（1）fun(a,low,high,k)算法的功能是在有序数组 a[low..high]中采用折半查找方法查找值为 k 的元素的下标，没有找到时返回－1。

（2）当 a[]={1,2,3,4,5,6,7,8}时，执行 fun(a,0,7,5)后的返回结果是 4。

5. 有一个递增有序表 R，以下算法利用有序性进行顺序查找：

```
int Find(RecType R[ ],int n,KeyType k)
{   int i = 0;
    while (i < n)
    {   if (R[i].key == k)
            return i + 1;
        else if (k > R[i].key)
            i++;
        else
            return 0;
    }
}
```

分析该算法在等概率情况下成功和不成功查找的平均查找长度，和单纯的顺序查找相比，哪个查找效率更高一些？

答：这是有序表上的顺序查找算法。设 $R=(a_0,a_1,\cdots,a_i,a_{i+1},\cdots,a_{n-1})$，在成功查找情况下，找到 a_i 元素，需要和 $a_0,a_1,\cdots,a_i$ 的元素进行比较，即比较 $i+1$ 次，所以：

$$\mathrm{ASL}_{\text{成功}}=\sum_{i=0}^{n-1}\frac{1}{n}(i+1)=\frac{n+1}{2}$$

在不成功查找情况下有 $n+1$ 种情况，其判定树如图 9.11 所示，则：

$$\mathrm{ASL}_{\text{不成功}}=\frac{1+2+\cdots+n+n}{n+1}=\frac{n}{2}+\frac{n+1}{n}$$

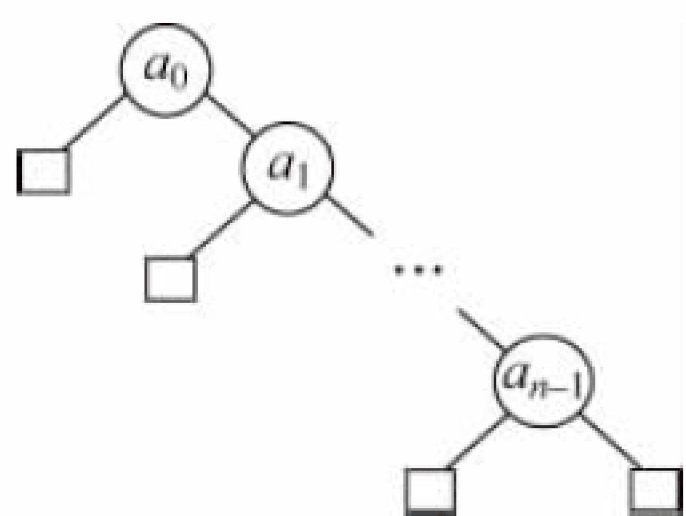


图 9.11 有序表上顺序查找的判定树

由于 $\mathrm{ASL}_{\text{不成功}}=\frac{n}{2}+\frac{n+1}{n}<n$（顺序查找算法的不成功查找长度为 n），两者成功查找的平均查找长度相同，所以本算法比单纯的顺序查找算法的查找效率更高一些。

6. 证明二叉排序树的中序遍历序列是从小到大有序的。

证明：采用反证法证明。

设二叉排序树的中序序列为 $R_1,R_2,R_3,\cdots,R_i,\cdots,R_j,\cdots,R_n$。并假设 $R_j<R_i$，根据二叉排序树的生成规则，R_i、R_j 一定是以某个结点 R_k 为根的子树中的结点，不妨设在生成二叉排序树时 R_i 先于 R_j 输入，而且 $R_i>R_k$（$R_i<R_k$ 的情况类似）。这样 R_i 输入后一定是 R_k 右子树中的结点。在 R_j 输入时，若 $R_j>R_k$，则 R_j 也成为 R_k 右子树的结点，但由于 $R_j<R_i$，所以 R_j 不可能成为 R_i 右子树中的结点。若 $R_j<R_k$，则 R_j 成为 R_k 左子树中的结点，这样按中序遍历所得的序列为：

$$\cdots,R_k,\cdots,R_j,\cdots,R_i,\cdots \quad 或 \quad \cdots,R_j,\cdots,R_k,\cdots,R_i,\cdots$$

这样与前面的中序序列矛盾，所以当 $R_j<R_i$ 时命题成立。

同理可以证明 R_j 先于 R_i 输入的情况。

7. 某人认为可以这样定义二叉排序树，对于一棵非空二叉树，若每个结点的关键字大于左孩子的关键字，小于右孩子的关键字，则它是一棵二叉排序树。如果你认为正确，请证明，如果你认为错误，请给出一个反例。

答：错误。反例如图 9.12 所示，它满足题目中的定义，但不是一棵二叉排序树。

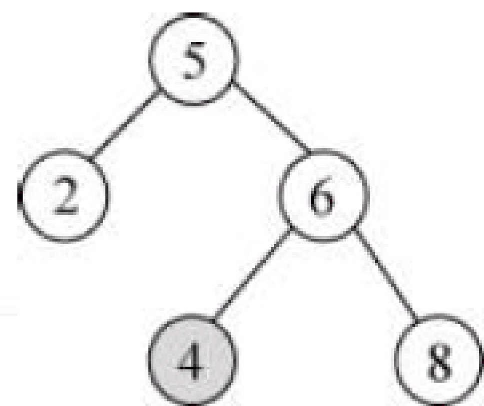


图 9.12 一棵非空二叉树

8. 给定一棵非空二叉排序树的先序序列，可以唯一确定该二叉排序树吗？为什么？

答：可以。由二叉排序树的先序序列可以确定其结点个数，将所有结点值递增排序构成其中序序列，由先序序列和中序序列可以唯一构造该二叉排序树。

9. 对于一棵高度为 10 的平衡二叉树，求叶子结点的最小层次。

答：设 L_h 是高度为 h 的平衡二叉树中叶子结点的最小层次，有以下关系。

$$L_1=1,\quad L_2=2,\quad L_h=\mathrm{MIN}\{L_{h-1},L_{h-2}\}+1（当\ h>2）$$

由此可以计算：

$$L_3 = \text{MIN}\{1,2\}+1=2,\quad L_4=\text{MIN}\{2,2\}+1=3,$$
$$L_5 = \text{MIN}\{3,2\}+1=3,\quad L_6=\text{MIN}\{3,3\}+1=4,$$
$$L_7 = \text{MIN}\{4,3\}+1=4,\quad L_8=\text{MIN}\{4,4\}+1=5,$$
$$L_9 = \text{MIN}\{5,4\}+1=5,\quad L_{10}=\text{MIN}\{5,5\}+1=6$$

由此可以推出 $L_h=\lfloor h/2 \rfloor+1$。

所以该平衡二叉树中叶子结点的最小层次为 6。

10. 试问含有 8 个关键字的 3 阶 B—树最多有几个内部结点？最少有几个内部结点？画出其形态。

答：3 阶 B—树每个结点的关键字个数为 1～2，除根结点外，所有结点均为一个关键字时结点最多，均为两个关键字时结点最少。所以含有 8 个关键字的 3 阶 B—树最多有 7 个内部结点，最少有 4 个内部结点，其形态分别如图 9.13(a)和(b)所示。其中，“•”表示一个关键字。

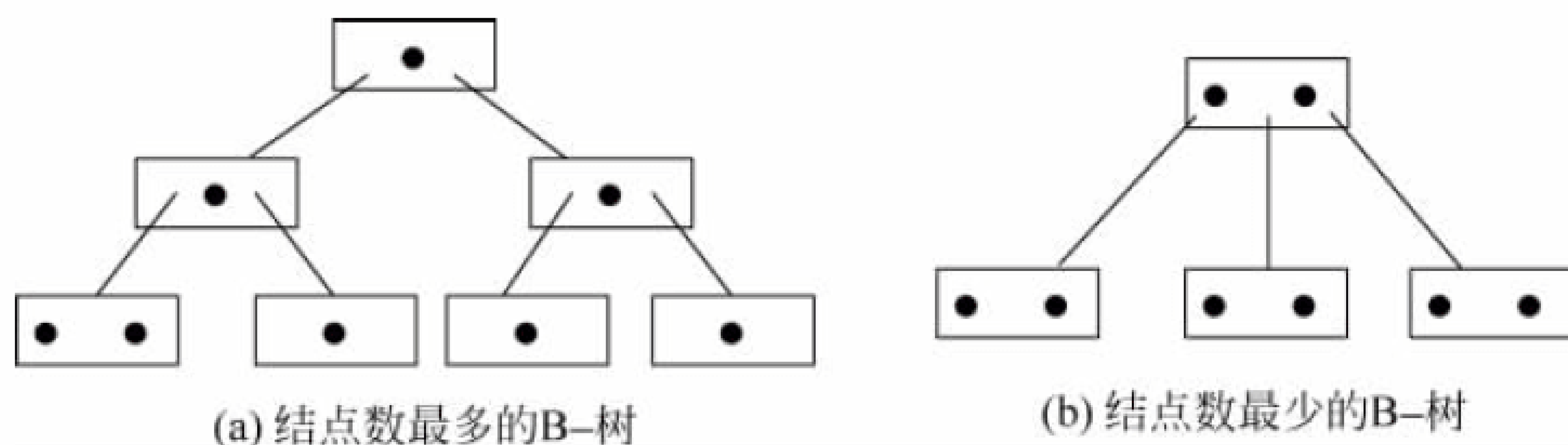


图 9.13　两棵含有 8 个关键字的 3 阶 B—树

11. 已知一棵 3 阶 B—树如图 9.14 所示，画出在其中插入关键字 18 的过程。

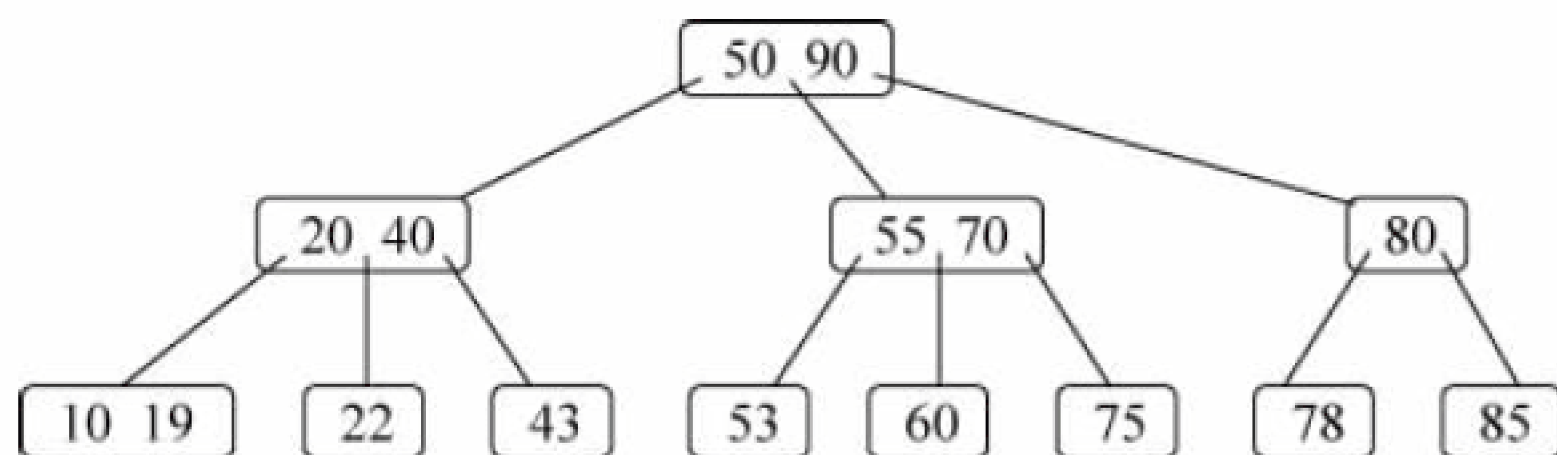


图 9.14　一棵 3 阶 B—树

答：3 阶 B—树每个结点的关键字个数为 1～2。在该 3 阶 B—树中插入 18，实际上插入到叶子结点(10，19)，该结点变为 10 18 19(关键字个数＞2)，如图 9.15(a)所示。分裂该结点，18 放入双亲结点，双亲结点变为 18 20 40(关键字个数＞2)，如图 9.15(b)所示。分裂该结点，20 放入双亲结点，双亲结点变为 20 50 90(关键字个数＞2)，如图 9.15(c)所示。分裂该结点，如图 9.15(d)所示。

12. 哈希表查找的时间性能在什么情况下可以达到 $O(1)$？

答：如果哈希函数设计得好，不出现同义词冲突现象，此时哈希表查找的时间性能可以达到 $O(1)$，这是一种理想情况。在存储的数据集是确定的情况下，并且关键字为整数，可以证明能够设计出这样的哈希函数。

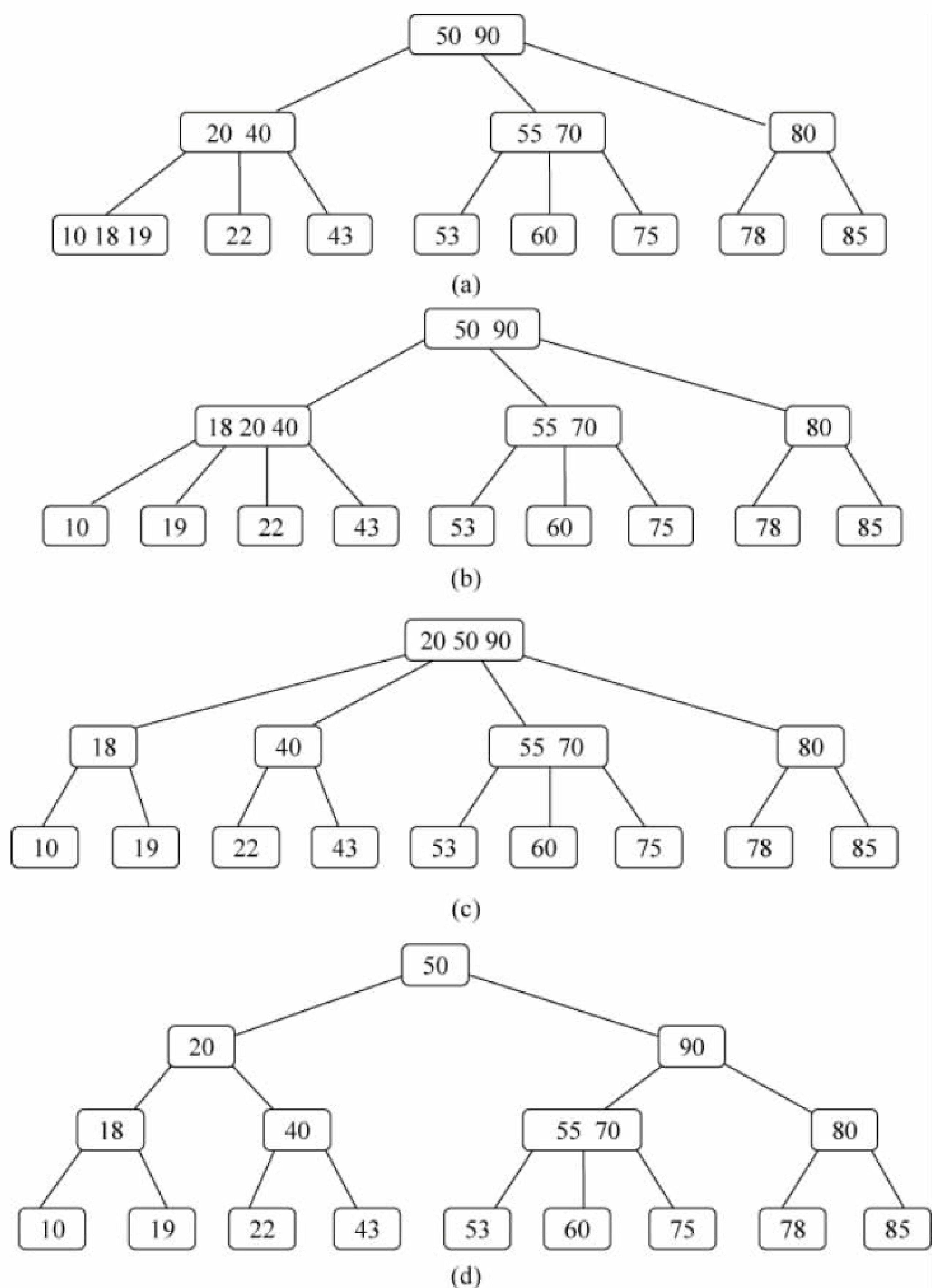


图 9.15 在 B—树中插入关键字 18 的过程

13. 为什么哈希表不支持元素之间的顺序查找？

答：哈希表是通过哈希地址来查找对应关键字的记录的，对哈希表来说顺序查找没有任何意义，也没有提供顺序查找机制。

14. 用开放定址法构造哈希表，其装填因子为何不能超过 1？而用拉链法构造哈希表，其装填因子为何可以超过 1？

答：用开放定址法构造哈希表，无论是否发生冲突，所有元素都放在同一个表内，因此表的空间要设计得足够大，一般不能装满，所以装填因子应小于 1。

用拉链法构造哈希表，每个哈希地址是一个链表的表头指针，所有元素另开辟空间存放，并未占用基本空间，表中元素个数可能大大超过表的大小，所以其装填因子可以超过 1。

15. 在采用线性探测法处理冲突的哈希表中，所有同义词在表中是否一定相邻？

答：不一定相邻。

在线性探测法中，哈希地址为 $i(0 \leqslant i \leqslant m-1)$ 的关键字和为了解决冲突形成的探测序列（即 i 的非同义词）都争夺哈希地址 i，所以同义词在表中不一定相邻。

16. 对于关键字序列(30,15,21,40,25,26,36,37)，若查找表的装填因子为0.8，采用线性探测法解决冲突，完成以下各题：

(1) 设计哈希函数。

(2) 画出哈希表。

(3) 计算在等概率条件下查找成功和查找失败的平均查找长度。

答：由于装填因子为0.8，关键字个数 $n=8$，所以表长 $m=8/0.8=10$。

(1) 用除留余数法，设哈希函数为 $H(\text{key})=\text{key} \bmod 7$。

(2) 设计的哈希表如表9.3所示。

表 9.3 一个哈希表

地址	0	1	2	3	4	5	6	7	8	9
关键字	21	15	30	36	25	40	26	37		
探测次数	1	1	1	3	1	1	2	6		

(3) 计算成功情况下的平均查找长度如下：

$$\text{ASL}_{\text{成功}} = \frac{1+1+1+3+1+1+2+6}{8} = 2$$

在查找失败时各 $H(\text{key})$（只能取0～6中的某个值）对应的探测次数如表9.4所示（仅考虑阴影部分），如某个关键字 k 查找失败，但 $H(k)=0$，需要9次比较才能确定查找失败。故查找失败时的平均查找长度为：

$$\text{ASL}_{\text{不成功}} = \frac{9+8+7+6+5+4+3}{7} = 6$$

表 9.4 失败时的探测次数

地址	0	1	2	3	4	5	6	7	8	9
关键字	21	15	30	36	25	40	26	37		
探测次数	9	8	7	6	5	4	3	2	1	1

17. 设有一组关键字{9,1,23,14,55,20,84,27}，采用哈希函数 $H(\text{key})=\text{key} \bmod 7$，表长 m 为10，用开放地址法的平方探测法 $H_i=(H(\text{key})+d_i) \bmod 10 (d_i=\pm 1^2, \pm 2^2, \pm 3^2, \cdots)$ 来解决冲突，要求对该关键字序列构造哈希表，并计算查找成功的平均查找长度。

答：设计的哈希表如表9.5所示。

表 9.5 一个哈希表

地址	0	1	2	3	4	5	6	7	8	9
关键字	14	1	9	23		27	55	20		84
探测次数	1	1	1	2		3	1	2		3

例如，对于关键字84：

$$H(84) = 84 \bmod 7 = 0（冲突）$$
$$H_1 = (0+1^2) \bmod 10 = 1（冲突）$$

$$H_2 = (0 - 1^2) \bmod 10 = 9$$

共探测了 3 次。

对于关键字 27：

$$H(27) = 27 \bmod 7 = 6(\text{冲突})$$
$$H_1 = (6 + 1^2) \bmod 10 = 7(\text{冲突})$$
$$H_2 = (6 - 1^2) \bmod 10 = 5$$

共探测了 3 次。

平均查找长度：$ASL_{成功} = \frac{1+1+1+2+3+1+2+3}{8} = 1.75$。

9.3.5 算法设计题

1.【顺序查找算法】设计一个顺序查找的递归算法。

解：对应的递归模型如下。

$f(R,n,k,i)=0$　　　　当 $i \geq n$

$f(R,n,k,i)=i+1$　　　　当 $R[i].\text{key}=k$

$f(R,n,k,i)=f(R,n,k,i+1)$　　　　其他情况

对应的算法如下：

```
int SeqSearch1(RecType R[],int n,KeyType k,int i)
//初始调用时 i=0
{   if (i>=n)
        return 0;
    else if (R[i].key==k)
        return i+1;
    else
        return(SeqSearch1(R,n,k,i+1));
}
```

2.【顺序查找算法】若顺序表中各元素的查找概率不等，可用以下策略提高顺序查找的效率：若找到指定的元素，则将该元素与其前驱(若存在)元素交换，使得经常被查找的元素尽量位于表的前端。设计出满足上述策略的顺序查找算法。

解：对应的算法如下。

```
int SeqSearch2(RecType R[],int n,KeyType k)
{   int i=0;
    while (i<n && R[i].key!=k)
        i++;
    if (i<n)                        //找到了关键字为 k 的元素 R[i]
    {   if (i>0)                    //i>0 时 R[i]前移一位
        {   swap(R[i], R[i-1]);     //R[i]交换到前面
            return i;               //返回 R[i]新位置的逻辑序号
        }
        else return i+1;            //i=0 时不能移动 R[i],返回 1
    }
```

```
    return 0;
}
```

3.【折半查找算法】设计一个折半查找的递归算法。

解：根据折半查找过程得到以下递归模型。

$$f(R,\text{low},\text{high},k)=\begin{cases}0 & \text{当 low} > \text{high 时}\\ \text{mid}+1(\text{mid}=(\text{low}+\text{high})/2) & \text{当 } R[\text{mid}].\text{key}=k \text{ 时}\\ f(R,\text{low},\text{mid}-1,k) & \text{当 } k<R[\text{mid}].\text{key 时}\\ f(R,\text{mid}+1,\text{high},k) & \text{当 } k>R[\text{mid}].\text{key 时}\end{cases}$$

对应的算法如下。

```
int BinSearch1(RecType R[],int low,int high,KeyType k)
{   //初始调用时 low = 0,high = n - 1
    int mid;
    if (low <= high)
    {   mid = (low + high)/2;
        if (R[mid].key < k)
            return BinSearch1(R,mid + 1,high,k);
        else if (R[mid].key > k)
            return BinSearch1(R,low,mid - 1,k);
        else                //R[mid].key = k
            return mid + 1;
    }
    else return 0;
}
```

4.【折半查找算法】利用折半查找算法在一个有序表 R 中插入一个关键字为 k 的元素 x，并保持表的有序性。

解：先采用折半查找算法找到插入元素的位置 pos，将位置 pos 及之后的所有元素后移一个位置，然后将 x 放置在位置 pos 处。对应的算法如下：

```
void Binsert(RecType R[],int &n,RecType x)
{   int low = 0,high = n - 1,mid,pos,i;
    bool find = false;
    while (low <= high && !find)     //折半查找
    {   mid = (low + high)/2;
        if (x.key < R[mid].key)
            high = mid - 1;
        else if (x.key > R[mid].key)
            low = mid + 1;
        else
        {   i = mid;
            find = true;
        }
    }
```

```
        if (find)                    //若找到相同关键字的元素
            pos = mid;               //在 mid 处插入元素
        else                         //否则有 x.key > R[high].key 并且 x.key < R[high + 1].key
            pos = high + 1;          //在该 high + 1 处插入元素
        for (i = n - 1;i >= pos;i -- )  //R[pos..n - 1]均后移一位
            R[i + 1] = R[i];
        R[pos] = x;                  //插入元素 x
        n++;                         //元素个数增加 1
    }
```

5.【二叉排序树算法】设计一个递归算法,从大到小输出二叉排序树 bt 中所有值不小于 k 的关键字。

解:由二叉排序树的性质可知,右子树中的所有结点值大于根结点值,左子树中的所有结点值小于根结点值。为了从大到小输出,先遍历右子树,再访问根结点(仅输出大于等于 k 的关键字),后遍历左子树。对应的算法如下:

```
void Output(BSTNode * bt, KeyType k)
{   if (bt == NULL)
        return;
    if (bt -> rchild!= NULL)
        Output(bt -> rchild,k);
    if (bt -> key >= k)
        printf(" %d ",bt -> key);
    if (bt -> lchild!= NULL)
        Output(bt -> lchild,k);
}
```

6.【二叉排序树算法】假设二叉排序树 bt 中所有的关键字是由整数构成的,为了查找某关键字 k,会得到一个查找序列。设计一个算法,判断一个序列(存放在 a 数组中)是否是从 bt 中搜索关键字 k 的序列。

解:设查找序列 a 中有 n 个关键字,如果查找成功,$a[n-1]$应等于 k。用 i 扫描 a 数组(初值为 0),p 用于在二叉排序树 bt 中查找(p 的初值指向根结点),每查找一层,比较该层的结点关键字 p->key 是否等于 $a[i]$,若不相等,表示 a 不是 bt 中查找关键字 k 的序列,返回 false,否则继续查找下去。若一直未找到关键字 k,则 p 最后必为 NULL,表示 a 不是查找序列,返回 false,否则表示在 bt 中查找到 k,p 指向该结点,表示 a 是查找序列,返回 true。对应的算法如下:

```
bool Findseq(BSTNode * bt, int k, int a[ ], int n)
{   BSTNode * p = bt;
    int i = 0;
    if (a[n - 1]!= k)                   //未找到 k,返回 false
        return false;
    while (i < n && p!= NULL)
    {   if (p -> key!= a[i])            //若不等,表示 a 不是 k 的查找序列
            return false;               //返回 false
```

```
        if (k<p->key) p=p->lchild;            //在左子树中查找
        else if (k>p->key) p=p->rchild;       //在右子树中查找
        i++;                                  //查找序列指向下一个关键字
    }
    if (p!=NULL) return true;                 //找到了 k,返回 true
    else return false;                        //未找到 k,返回 false
}
```

7.【二叉排序树算法】对于二叉排序树 bt,设计一个算法,删除其中以关键字 k 为根结点的子树中所有关键字小于 k 的结点。

解:利用二叉排序树销毁算法 DestroyBST(bt),它用于删除并释放以 bt 为根结点的二叉排序树。在二叉排序树 bt 中找关键字为 k 的结点 bt,调用 DestroyBST(bt->lchild)删除以 bt 为根结点的左子树,并置 bt->lchild 为空。对应的算法如下:

```
bool Delk(BSTNode *&bt,KeyType k)
{   if (bt==NULL)
        return false;
    if (bt->key==k)
    {   DestroyBST(bt->lchild);
        bt->lchild=NULL;
        return true;
    }
    else if (k<bt->key)
        Delk(bt->lchild,k);
    else
        Delk(bt->rchild,k);
}
```

8.【二叉排序树算法】对于二叉排序树 bt,设计一个算法,输出在该树中查找某个关键字 k 所经过的路径。

解:设计的算法为 SearchPath(BSTNode * bt,KeyType k,KeyType path[],int d),它输出二叉排序树 bt 中查找关键字 k 的查找路径。使用 path 数组存储经过的结点,d 表示 path 中最后关键字的下标,初始值为−1。当找到所要找的结点时,输出 path 数组中的元素值,从而以根结点到当前结点输出路径。对应的算法如下:

```
void SearchPath(BSTNode *bt,KeyType k,KeyType path[],int d)
{   //d 表示 path 中最后关键字的下标,初始值为-1
    if (bt==NULL)
        return;
    else if (k==bt->key)
    {   d++; path[d]=bt->key;                 //添加到路径中
        for (int i=0;i<=d;i++)
            printf("%2d",path[i]);
        printf("\n");
    }
    else
```

```
	{	d++; path[d] = bt -> key;			//添加到路径中
		if (k < bt -> key)
			SearchPath(bt -> lchild,k,path,d);	//在左子树中递归查找
		else
			SearchPath(bt -> rchild,k,path,d);	//在右子树中递归查找
	}
}
```

9.【平衡二叉树算法】设计一个算法，判断一棵二叉排序树 bt 是否为平衡的。

解：对于二叉排序树 bt，用 balance 表示其平衡性，h 表示二叉排序树 bt 的高度。对于 bt 结点，递归调用 JudgeAVT(bt—>lchild,bl,hl)求出左子树的平衡性 bl 和高度 hl，递归调用 JudgeAVT(bt—>rchild,br,hr)求出右子树的平衡性 br 和高度 hr，若 hl 和 hr 之差的绝对值小于等于 1，说明该结点是平衡的，返回左、右子树的平衡性，即 bl & br，否则表示是不平衡的。对应的算法如下：

```
int abs(int x,int y)				//求两个整数差的绝对值
{	int z = x - y;
	return z > 0?z: - z;
}
void JudgeAVT(BSTNode * bt,bool &balance,int &h)
{	//h 为 bt 的高度,balance 表示 bt 的平衡性
	int hl,hr;
	bool bl,br;
	if (bt == NULL)				//空树是平衡的
	{	h = 0;
		balance = true;
	}
	else if (bt -> lchild == NULL && bt -> rchild == NULL)	//只有一个结点的树是平衡的
	{	h = 1;
		balance = true;
	}
	else
	{	JudgeAVT(bt -> lchild,bl,hl);	//求出左子树的平衡性 bl 和高度 hl
		JudgeAVT(bt -> rchild,br,hr);	//求出右子树的平衡性 br 和高度 hr
		h = (hl > hr?hl:hr) + 1;
		if (abs(hl,hr)<= 1)
			balance = bl & br;		//& 为逻辑与运算符
		else
			balance = false;
	}
}
```

设计以下主函数：

```
int main()
{	BSTNode * bt;
	KeyType keys[] = {5,2,3,4,1,8,6,7,9};
	int n = 9;
```

```
    bt = CreateBST(keys,n);            //创建二叉排序树
    printf("BST:"); DispBST(bt); printf("\n");
    bool balance;
    int h;
    JudgeAVT(bt,balance,h);
    if (balance)
        printf("该 BST 树是平衡的\n");
    else
        printf("该 BST 树是不平衡的\n");
    printf("该 BST 树的高度: %d\n",h);
    DestroyBST(bt);
    return 1;
}
```

程序的执行结果如下：

```
BST:5(2(1,3(,4)),8(6(,7),9))
该 BST 树是平衡的
该 BST 树的高度:4
```

第 10 章 内排序



10.1 本章知识体系



1. 知识结构图

本章的知识结构如图 10.1 所示。

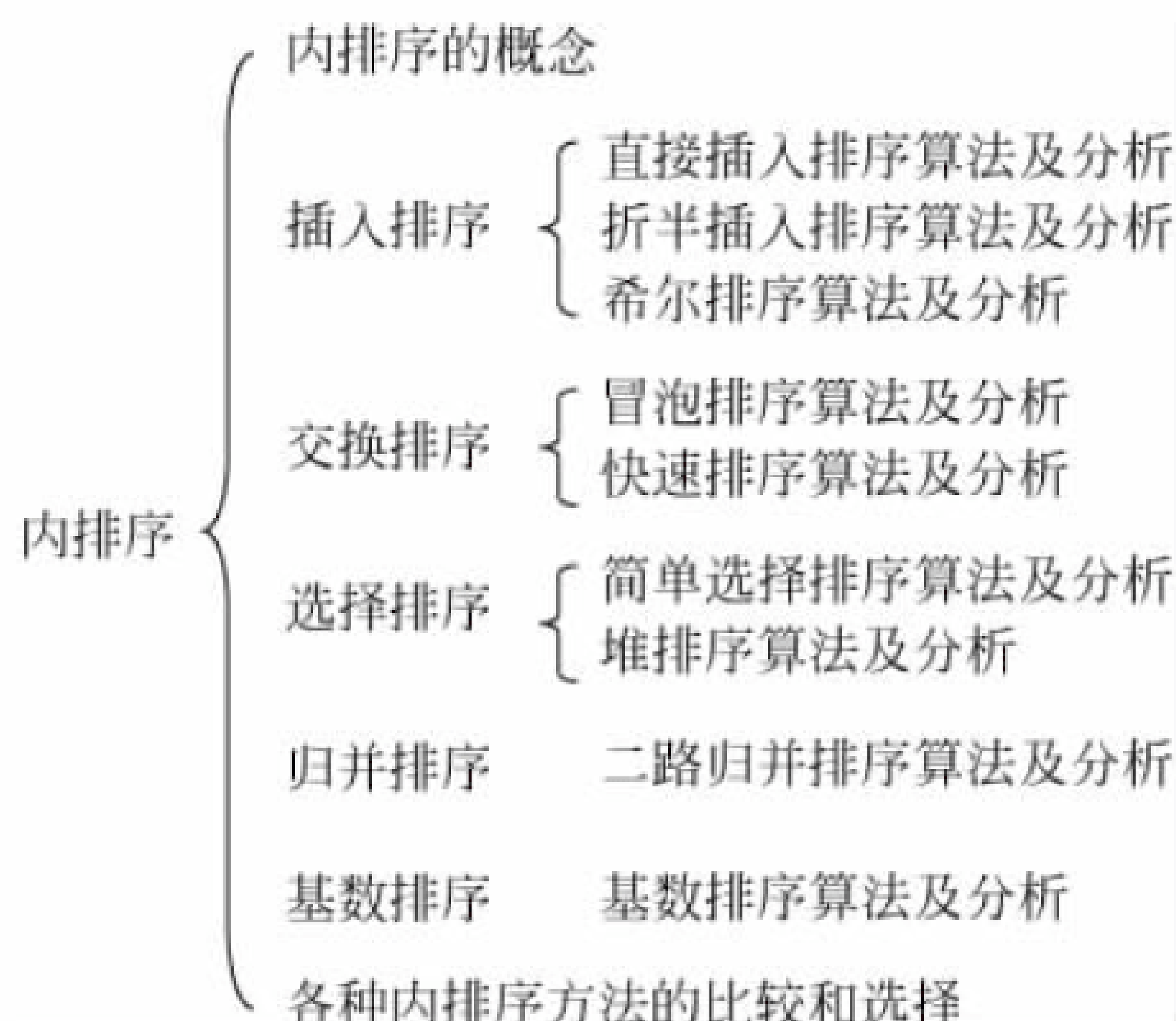


图 10.1　第 10 章知识结构图

2. 基本知识点

(1) 内排序相关概念和排序算法分析方法。

(2) 简单类排序算法(直接插入、冒泡排序和简单选择排序)设计。

(3) 折半插入排序算法设计。

(4) 希尔排序算法设计。

(5) 快速排序算法设计。

(6) 堆排序算法设计。

(7) 二路归并排序算法设计。

(8) 基数排序算法设计。

(9) 各种内排序算法的特点及其应用。

3. 要点归纳

(1) 稳定的排序算法是指多个相同关键字记录在排序后相对位置不发生改变。

(2) 内排序的主要时间花在关键字的比较和记录的移动上。

(3) 直接插入排序在初始数据正序时呈现最好情况,此时时间复杂度为 $O(n)$;在初始数据反序时呈现最坏情况,此时时间复杂度为 $O(n^2)$。平均情况接近最坏情况。

(4) 直接插入排序和折半插入排序每趟产生的有序区是局部有序区。

(5) 折半插入排序采用折半查找方法找插入记录的位置,但和直接插入排序移动记录的次数是相同的。

(6) 希尔排序是利用了直接插入排序在数据正序时呈现最好情况这个特点。

(7) 冒泡排序在初始数据正序时呈现最好情况,此时时间复杂度为 $O(n)$;在初始数据反序时呈现最坏情况,此时时间复杂度为 $O(n^2)$。平均情况接近最坏情况。

(8) 冒泡排序每趟产生的有序区是全部有序区。每趟将一个记录归位,以后该记录位置不再发生改变。

(9) 快速排序利用划分方法实现排序,将一个无序区分为两个子无序区,每个子无序区的处理是独立的,哪个先处理都可以。

(10) 快速排序每趟将一个记录归位,以后该记录位置不再发生改变,但每趟并不产生有序区。

(11) 快速排序在初始数据正序和反序时都呈现最坏情况,而平均情况接近最好情况。快速排序的空间复杂度为 $O(\log_2 n)$。

(12) 选择类排序算法(包括简单选择排序和堆排序)与初始数据的正反序无关,它们都是不稳定的。

(13) 简单选择排序采用简单比较方法从 k 个记录中挑选出最大(或者最小)关键字记录,执行时间为 $O(k)$。而堆排序是采用堆结构来实现的,执行时间为 $O(\log_2 k)$。

(14) 简单选择排序的平均情况接近最坏情况,而堆排序的平均情况接近最好情况。

(15) 在一个大根堆中,根结点是关键字最大的结点,关键字最小的结点一定属于叶子结点。从根结点到某个叶子结点的路径恰好构成一个关键字递减序列。

(16) 堆的用途是挑选最大(或者最小)关键字记录,当需要从一组记录中连续挑选最大(或者最小)关键字记录时可以采用堆来实现。对一个堆进行层次遍历,不一定能得到一个有序序列。

(17) 二路归并排序的空间复杂度为 $O(n)$,它是本课程介绍的各种内排序中空间复杂度最高的排序方法。

(18) 三路归并排序的时间复杂度也是 $O(n\log_2 n)$,但其算法设计远比二路归并排序算法复杂。

(19) 基数排序不需要关键字比较,其空间复杂度为 $O(r)$。

(20) 基数排序的每一趟都是稳定的,所以基数排序是稳定的排序方法。

(21) 在多个关键字排序中可能需要考虑排序算法的稳定性。

(22) 不能简单地认为一种排序方法就一定好于另外一种排序方法,影响排序性能的因素有多个。

10.2 教材中的练习题及参考答案

1. 直接插入排序算法在含有 n 个元素的初始数据正序、反序和数据全部相等时时间复杂度各是多少?

答:含有 n 个元素的初始数据正序时,直接插入排序算法的时间复杂度为 $O(n)$。

含有 n 个元素的初始数据反序时,直接插入排序算法的时间复杂度为 $O(n^2)$。

含有 n 个元素的初始数据全部相等时,直接插入排序算法的时间复杂度为 $O(n)$。

2. 回答以下关于直接插入排序和折半插入排序的问题：

(1) 使用折半插入排序所要进行的关键字比较次数是否与待排序的元素的初始状态有关？

(2) 在一些特殊情况下，折半插入排序比直接插入排序要执行更多的关键字比较，这句话对吗？

答：(1) 使用折半插入排序所要进行的关键字比较次数与待排序的元素的初始状态无关。无论待排序序列是否有序，已形成的部分子序列是有序的。折半插入排序首先查找插入位置，插入位置是判定树失败的位置(对应外部结点)，失败位置只能在判定树的最下两层上。

(2) 在一些特殊情况下，折半插入排序的确比直接插入排序需要更多的关键字比较，例如在待排序序列正序的情况下便是如此。

3. 有以下关于排序的算法：

```
void fun(int a[ ],int n)
{   int i,j,d,tmp;
    d = n/3;
    while (true)
    {   for (i = d;i < n;i++)
        {   tmp = a[i];
            j = i - d;
            while (j >= 0 && tmp < a[j])
            {   a[j + d] = a[j];
                j = j - d;
            }
            a[j + d] = tmp;
        }
        if (d == 1) break;
        else if (d < 3) d = 1;
        else d = d/3;
    }
}
```

(1) 指出 fun(a,n)算法的功能。

(2) 当 a[]={5,1,3,6,2,7,4,8}时，问 fun(a,8)共执行几趟排序？各趟的排序结果是什么？

答：(1) fun(a,n)算法的功能是采用增量递减为 1/3 的希尔排序方法对 a 数组中的元素进行递增排序。

(2) 当 a[]={5,1,3,6,2,7,4,8}时，执行 fun(a,8)的过程如下。

d=2：2 1 3 6 4 7 5 8

d=1：1 2 3 4 5 6 7 8

共有两趟排序。

4. 在实现快速排序的非递归算法时，可根据基准元素将待排序序列划分为两个子序列。若下一趟首先对较短的子序列进行排序，试证明在此做法下快速排序所需要的栈的深

度为 $O(\log_2 n)$。

答：由快速排序的算法可知，所需递归工作栈的深度取决于所需划分的最大次数。在排序过程中每次划分都把整个待排序序列根据基准元素划分为左、右两个子序列，然后对这两个子序列进行类似的处理。设 $S(n)$ 为对 n 个记录进行快速排序时平均所需栈的深度，则：

$$S(n)=\frac{1}{n}\sum_{k=1}^{n}(S(k-1)+S(n-k))=\frac{2}{n}\sum_{i=0}^{n-1}S(i)$$

当 $n=1$ 时，所需栈空间为常量，由此可推出 $S(n)=O(\log_2 n)$。

实际上，在快速排序中下一趟首先对较短子序列排序，并不会改变所需栈的深度，所以所需栈的深度仍为 $O(\log_2 n)$。

5. 将快速排序算法改为非递归算法时通常使用一个栈，若把栈换为队列会对最终排序结果有什么影响？

答：在执行快速排序算法时，用栈保存每趟快速排序划分后左、右子区间的首、尾地址，其目的是为了在处理子区间时能够知道其范围，这样才能对该子区间进行排序，但这与处理子区间的先后顺序没什么关系，而仅仅起存储作用(因为左、右子区间的处理是独立的)。因此，用队列同样可以存储子区间的首、尾地址，即可以取代栈的作用。在执行快速排序算法时，把栈换为队列对最终排序结果不会产生任何影响。

6. 在堆排序、快速排序和二路归并排序中：

(1) 若只从存储空间考虑，应首先选取哪种排序方法？其次选取哪种排序方法？最后选取哪种排序方法？

(2) 若只从排序结果的稳定性考虑，则应选取哪种排序方法？

(3) 若只从最坏情况下的排序时间考虑，则不应选取哪种排序方法？

答：(1) 若只从存储空间考虑，则应首先选取堆排序(空间复杂度为 $O(1)$)，其次选取快速排序(空间复杂度为 $O(\log_2 n)$)，最后选取二路归并排序(空间复杂度为 $O(n)$)。

(2) 若只从排序结果的稳定性考虑，则应选取二路归并排序，其他两种排序方法是不稳定的。

(3) 若只从最坏情况下的排序时间考虑，则不应选取快速排序方法。因为快速排序方法的最坏情况下的时间复杂度为 $O(n^2)$，其他两种排序方法在最坏情况下的时间复杂度为 $O(n\log_2 n)$。

7. 如果只想在一个有 n 个元素的任意序列中得到其中最小的第 $k(k\ll n)$ 个元素之前的部分排序序列，那么最好采用什么排序方法？为什么？例如有一个序列(57,40,38,11,13,34,48,75,6,19,9,7)，要得到其第 4 个元素($k=4$)之前的部分有序序列，用所选择的算法实现时要执行多少次比较？

答：采用堆排序最合适，建立初始堆(小根堆)所花的时间不超过 $4n$，每次选出一个最小元素所花的时间为 $\log_2 n$，因此得到第 k 个最小元素之前的部分序列所花的时间大约为 $4n+k\log_2 n$，而冒泡排序和简单选择排序所花的时间为 kn。

对于序列(57,40,38,11,13,34,48,75,6,19,9,7)，形成初始堆(小根堆)并选最小元素6，需进行 18 次比较；选次小元素 7 时，需进行 5 次比较；再选元素 9 时，需进行 6 次比较；选元素 11 时，需进行 4 次比较，总共需进行 33 次比较。整个过程如图 10.2 所示。

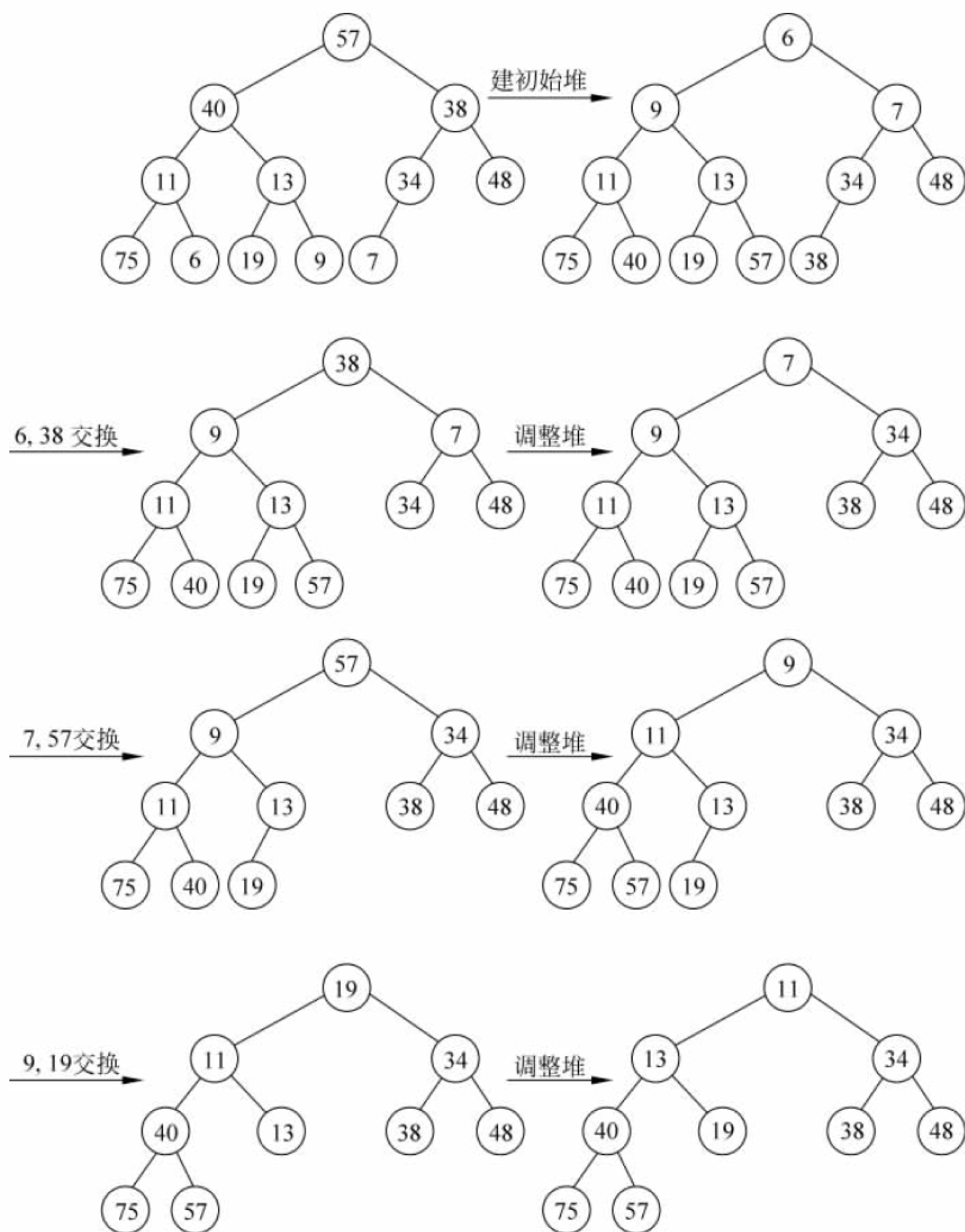


图 10.2 建立初始堆和产生部分有序序列的过程

8. 在基数排序过程中用队列暂存排序的元素，是否可以用栈来代替队列？为什么？

答：不能用栈来代替队列。

基数排序是一趟一趟进行的，从第 2 趟开始必须采用稳定的排序方法，否则排序结果可能不正确，若用栈代替队列，这样可能使排序过程变得不稳定。

9. 线性表有顺序表和链表两种存储方式，不同的排序方法适合不同的存储结构。对于常见的内部排序方法，说明哪些更适合于顺序表？哪些更适合于链表？哪些两者都适合？

答：更适合于顺序表的排序方法有希尔排序、折半插入排序、快速排序、堆排序和归并排序。

更适合于链表的排序方法是基数排序。

两者都适合的排序方法有直接插入排序、冒泡排序和简单选择排序。

10. 设一个整数数组 $a[0..n-1]$ 中存有互不相同的 n 个整数，且每个元素的值均在

1～n 之间。设计一个算法在 $O(n)$ 时间内将 a 中的元素递增排序，将排序结果放在另一个同样大小的数组 b 中。

解：对应的算法如下。

```
void fun(int a[],int n,int b[])
{   int i;
    for (i=0;i<n;i++)
        b[a[i]-1]=a[i];
}
```

11. 设计一个双向冒泡排序的算法，即在排序过程中交替改变扫描方向。

解：置 i 的初值为 0，先从后向前从无序区 $R[i..n-i-1]$ 归位一个最小元素 $R[i]$ 到有序区 $R[0..i-1]$，再从前向后从无序区 $R[i..n-i-1]$ 归位一个最大元素到有序区 $R[n-i..n-1]$。当某趟没有元素交换时结束，否则将 i 增加 1。对应的算法如下：

```
void DBubbleSort(RecType R[],int n)          //对R[0..n-1]按递增序进行双向冒泡排序
{   int i=0,j;
    bool exchange=true;                      //exchange标识本趟是否进行了元素交换
    while (exchange)
    {   exchange=false;
        for (j=n-i-1;j>i;j--)
            if (R[j].key<R[j-1].key)         //由后向前冒泡小元素
            {   exchange=true;
                swap(R[j],R[j-1]);           //R[j]和R[j-1]交换
            }
        for (j=i;j<n-i-1;j++)
            if (R[j].key>R[j+1].key)         //由前向后冒泡大元素
            {   exchange=true;
                swap(R[j],R[j+1]);           //R[j]和R[j+1]交换
            }
        if (!exchange) return;
        i++;
    }
}
```

12. 假设有 n 个关键字不同的记录存于顺序表中，要求不经过整体排序从中选出从大到小顺序的前 $m(m \ll n)$ 个元素。试采用简单选择排序算法实现此选择过程。

解：改进后的简单选择排序算法如下。

```
void SelectSort1(RecType R[],int n,int m)
{   int i,j,k;
    for (i=0;i<m;i++)                   //做第i趟排序
    {   k=i;
        for (j=i+1;j<n;j++)             //在当前无序区R[i..n-1]中选key最大的R[k]
            if (R[j].key>R[k].key)
                k=j;                    //k记下目前找到的最大关键字所在的位置
        if (k!=i)
```

```
            swap(R[i],R[k]);        //交换 R[i]和 R[k]
        }
    }
```

13. 对于给定的含有 n 个元素的无序数据序列(所有元素的关键字不相同),利用快速排序方法求这个序列中第 $k(1\leqslant k\leqslant n)$小元素的关键字,并分析所设计算法的最好和平均时间复杂度。

解:采用快速排序思想求解,当划分的基准元素为 $R[i]$时,根据 i 与 k 的大小关系在相应的子区间中查找。对应的算法如下:

```
KeyType QuickSelect(RecType R[ ],int s,int t,int k)  //在 R[s..t]序列中找第 k 小的元素
{   int i = s,j = t;
    RecType tmp;
    if (s<t)                              //区间内至少存在两个元素的情况
    {   tmp = R[s];                       //用区间的第 1 个记录作为基准
        while (i!= j)                     //从区间两端交替向中间扫描,直到 i = j 为止
        {   while (j > i && R[j].key >= tmp.key)
                j -- ;                    //从右向左扫描,找第 1 个关键字小于 tmp 的 R[j]
            R[i] = R[j];                  //将 R[j]前移到 R[i]的位置
            while (i < j && R[i].key <= tmp.key)
                i++;                      //从左向右扫描,找第 1 个关键字大于 tmp 的 R[i]
            R[j] = R[i];                  //将 R[i]后移到 R[j]的位置
        }
        R[i] = tmp;
        if (k - 1 == i) return R[i].key;
        else if (k - 1 < i) return QuickSelect(R,s,i - 1,k);     //在左区间中递归查找
        else return QuickSelect(R,i + 1,t,k); //在右区间中递归查找
    }
    else if (s == t && s == k - 1)        //区间内只有一个元素且为 R[k - 1]
        return R[k - 1].key;
    else
        return - 1;                       //k 错误返回特殊值 - 1
}
```

对于 QuickSelect(R,s,t,k)算法,设序列 R 中含有 n 个元素,其比较次数的递推式为:

$$T(n) = T(n/2) + O(n)$$

可以推导出 $T(n)=O(n)$,这是最好的情况,即每次划分的基准恰好是中位数,将一个序列划分为长度大致相等的两个子序列。在最坏情况下,每次划分的基准恰好是序列中的最大值或最小值,则处理区间只比上一次减少一个元素,此时比较次数为 $O(n^2)$。在平均情况下该算法的时间复杂度为 $O(n)$。

14. 设 n 个记录 $R[0..n-1]$的关键字只取 3 个值,即 0、1、2,采用基数排序方法将这 n 个记录排序,并用相关数据进行测试。

解:采用基数排序法,将关键字为 3 个值的记录分别放到 3 个队列中,然后收集起来即可。对应的算法如下:

```
#include "seqlist.cpp"                              //顺序表的基本运算算法
#include <malloc.h>
#define Max 3
typedef struct node
{   RecType Rec;
    struct node *next;
} NodeType;
void RadixSort1(RecType R[],int n)
{   NodeType *head[Max], *tail[Max], *p, *t;       //定义各链队的首、尾指针
    int i,k;
    for (i=0;i<Max;i++)                             //初始化各链队的首、尾指针
        head[i]=tail[i]=NULL;
    for (i=0;i<n;i++)
    {   p=(NodeType *)malloc(sizeof(NodeType)); //创建新结点
        p->Rec=R[i];
        p->next=NULL;
        k=R[i].key;                                 //找第 k 个链队,k=0,1 或 2
        if (head[k]==NULL)                          //进行分配,采用前插法建表
        {   head[k]=p; tail[k]=p; }
        else
        {   tail[k]->next=p; tail[k]=p; }
    }
    p=NULL;
    for (i=0;i<Max;i++)                             //对于每一个链队进行循环收集
        if (head[i]!=NULL)                          //产生以 p 为首结点指针的单链表
        {   if (p==NULL)
            {   p=head[i]; t=tail[i]; }
            else
            {   t->next=head[i]; t=tail[i];    }
        }
        i=0;
        while (p!=NULL)                             //将排序后的结果放到 R[]数组中
        {   R[i++]=p->Rec;
            p=p->next;
        }
}
```

设计以下主函数:

```
int main()
{   int i,n=5;
    RecType R[MAXL]={{1,'A'},{0,'B'},{0,'C'},{2,'D'},{1,'F'}};
    printf("排序前:\n  ");
    for (i=0;i<n;i++)
        printf("[%d,%c] ",R[i].key,R[i].data);
    printf("\n");
    RadixSort1(R,n);
    printf("排序后:\n  ");
    for (i=0;i<n;i++)
```

```
        printf("[%d,%c] ",R[i].key,R[i].data);
    printf("\n");
    return 1;
}
```

程序的执行结果如下：

```
排序前:
  [1,A]  [0,B]  [0,C]  [2,D]  [1,F]
排序后:
  [0,B]  [0,C]  [1,A]  [1,F]  [2,D]
```

显然，RadixSort1()算法的时间复杂度为 $O(n)$。

10.3 补充练习题及参考答案



10.3.1 单项选择题

1. 在下列排序方法中，执行时间不受数据初始状态影响，总为 $O(n\log_2 n)$ 的是________。

   A. 堆排序　B. 冒泡排序　C. 简单选择排序　D. 快速排序

答：A。

2. 在下列排序方法中，某一趟结束后未必能选出一个元素放在其最终位置上的是________。

   A. 堆排序　B. 冒泡排序　C. 直接插入排序　D. 快速排序

答：直接插入排序每趟产生的有序区是局部的，而快速排序每次划分归位一个元素，其他两种排序方法每趟产生的有序区是全局的。本题的答案为 C。

3. 在下列排序方法中，若待排序的数据已经有序，花费时间反而最多的是________。

   A. 快速排序　B. 希尔排序　C. 冒泡排序　D. 堆排序

答：A。

4. 在以下排序方法中，最耗费内存的是________。

   A. 快速排序　B. 堆排序
   C. 二路归并排序　D. 直接插入排序

答：C。

5. 对同一个排序序列分别进行折半插入排序和直接插入排序，两者之间可能的不同之处是________。

   A. 排序的总趟数　B. 元素的移动次数
   C. 使用辅助空间的数量　D. 元素之间的比较次数

答：折半插入排序采用折半查找方法找插入元素的位置，当 n 比较大时，元素之间的比较次数相对较少。本题的答案为 D。

6. 数据序列(8,9,10,4,5,6,20,1,2)只能是________算法的两趟排序后的结果。

A. 简单选择排序　B. 冒泡排序　C. 直接插入排序　D. 堆排序

答：采用排除法，该两趟排序后结果中的有序区不是全局有序的，所以只能是直接插入排序，不可能是其他3种排序方法。本题的答案为C。

7. 对数据序列(15,9,7,8,20,−1,4)进行排序，进行一趟后数据的排序变为(4,9,−1,8,20,7,15)，则采用的是________算法。

A. 简单选择排序　B. 冒泡排序　C. 希尔排序　D. 快速排序

答：使用各种排序方法对(15,9,7,8,20,−1,4)进行排序，看出是希尔排序。故本题的答案为C。

8. 依次将待排序序列中的元素插入到有序子序列中并扩大有序子序列的排序方法是________。

A. 快速排序　B. 直接插入排序　C. 冒泡排序　D. 堆排序

答：该排序方法主要体现在插入操作上。本题的答案为B。

9. 若表 R 的初始数据接近正序排列，则________方法的比较次数最少。

A. 直接插入排序　B. 快速排序　C. 归并排序　D. 简单选择排序

答：A。

10. 已知表 R 中的每个元素距其最终位置不远，采用________方法最节省时间。

A. 堆排序　B. 直接插入排序　C. 快速排序　D. 简单选择排序

答：表 R 中的每个元素距其最终位置不远，表示接近正序。本题的答案为B。

11. 在下列排序方法中，关键字比较的次数与记录的初始排列次序无关的是________。

A. 希尔排序　B. 冒泡排序

C. 直接插入排序　D. 简单选择排序

答：选择类排序方法的执行时间复杂度与记录的初始排列次序无关。本题的答案为D。

12. 快速排序方法在________情况下最不利于发挥其长处。

A. 要排序的数据量太大　B. 要排序的数据中含有多个相同值

C. 要排序的数据已基本有序　D. 要排序的数据个数为奇数

答：C。

13. 若 R 中有10 000个元素，如果仅要求求出其中最大的10个元素，则采用________方法最节省时间。

A. 堆排序　B. 希尔排序　C. 快速排序　D. 基数排序

答：堆排序每次输出一个堆顶元素(即最大或最少值的元素)，然后对堆进行调整，又可以挑选出次最大或次最小元素。本题的答案为A。

14. 内排序方法的稳定性是指________。

A. 该排序算法不允许有相同的关键字记录

B. 该排序算法允许有相同的关键字记录

C. 平均时间为 $O(n\log_2 n)$ 的排序方法

D. 以上都不对

答：D。

15. 在以下各排序方法中，________是不稳定的排序方法。

A. 直接插入排序　B. 冒泡排序　C. 二路归并排序　D. 堆排序

答：D。

16. 在以下各排序方法中，________是稳定的排序方法。

A. 直接插入和快速排序　B. 快速排序和堆排序

C. 简单选择和二路归并排序　D. 二路归并排序和冒泡排序

答：D。

17. 在以下各排序方法中，________是稳定的排序方法。

A. 简单选择排序　B. 折半插入排序　C. 希尔排序　D. 快速排序

答：B。

18. 若需在 $O(n\log_2 n)$的平均时间内完成对顺序表的排序，且要求排序是稳定的，则可选择的排序方法是________。

A. 快速排序　B. 堆排序　C. 二路归并排序　D. 直接插入排序

答：C。

19. 在以下各排序方法中，辅助空间为 $O(n)$的是________。

A. 堆排序　B. 二路归并排序　C. 希尔排序　D. 快速排序

答：B。

20. 若一组记录的关键字序列为(46,79,56,38,40,84)，利用堆排序的方法建立的初始堆为________。

A. 79,46,56,38,40,80　B. 84,79,56,38,40,46

C. 84,79,56,46,40,38　D. 84,56,79,40,46,38

答：B。

21. 若一组记录的关键字序列为(46,79,56,38,40,84)，则利用快速排序的方法，以第1个记录为基准得到的一次划分结果为________。

A. 38,40,46,56,79,84　B. 40,38,46,79,56,84

C. 40,38,46,56,79,84　D. 40,38,46,84,56,79

答：C。

22. 一组记录的关键字序列为(25,48,16,35,79,82,23,40,36,72)，其中含有5个长度为2的有序表，按二路归并排序方法对该序列再进行一趟归并后的结果为________。

A. 16,25,35,48,23,40,79,82,36,72

B. 16,25,35,48,79,82,23,36,40,72

C. 16,25,48,35,79,82,23,36,40,72

D. 16,25,35,48,79,23,36,40,72,82

答：(25,48,16,35,79,82,23,40,36,72)是 length＝2 的排序结果，下一趟归并取 length＝4。本题的答案为 A。

23. 已知10个数据元素为(54,28,16,34,73,62,95,60,26,43)，对该数列按从小到大排序，经过一趟冒泡排序后的序列为________。

A. 16,28,34,54,73,62,60,26,43,95

B. 28,16,34,54,62,73,60,26,43,95

C. 28,16,34,54,62,60,73,26,43,95

D. 16,28,34,54,62,60,73,26,43,95

答：冒泡排序每趟经过比较交换从无序区中产生一个最大的元素，所以答案为 B。

24. 用某种排序方法对线性表(25,84,21,47,15,27,68,35,20)进行排序时元素序列的变化情况如下：

(1) 25,84,21,47,15,27,35,68,20

(2) 21,25,47,84,15,27,35,68,20

(3) 15,21,25,27,35,47,68,84,20

(4) 15,20,21,25,27,35,47,68,84

其所采用的排序方法是________。

A. 简单选择排序　　B. 希尔排序

C. 二路归并排序　　D. 快速排序

答：C。

25. 有一组序列(48,36,68,99,75,24,28,52)进行快速排序，要求结果从小到大排序，则进行一次划分之后结果为________。

A. [24 28 36] 48 [52 68 75 99]　　B. [28 36 24] 48 [75 99 68 52]

C. [36 88 99] 48 [75 24 28 52]　　D. [28 36 24] 48 [99 75 68 52]

答：B。

26. 在以下排序方法中，最好情况下时间复杂度为 $O(n)$ 的依次是__①__、__②__。

A. 直接插入排序　　B. 简单选择排序

C. 冒泡排序　　D. 快速排序

答：①A　②C。

27. 在以下排序方法中，最坏情况下时间复杂度为 $O(n^2)$ 的依次是__①__、__②__。

A. 直接插入排序　　B. 简单选择排序

C. 堆排序　　D. 二路归并排序

答：① A　② B。

28. 在以下排序方法中，平均时间复杂度为 $O(n^2)$ 的依次是__①__、__②__。

A. 直接插入排序　　B. 冒泡排序

C. 二路归并排序　　D. 基数排序

答：①A　②B。

10.3.2 填空题

1. 若不考虑基数排序，则在其他几种内排序方法中主要进行的两种基本操作是关键字的__①__和记录的__②__。

答：① 比较　② 移动。

2. 对一组数据(4,48,96,23,12,60,45,73)采用直接插入排序算法进行递增排序，当把 60 插入到有序表中时，为寻找插入位置需比较________次。

答：2。插入 60 之前的结果是(4,12,23,48,96,60,45,73)，将 60 与 96、48 比较找到插入位置。

3. 对一组数据(4,48,96,23,12,60,45,73)采用折半插入排序算法进行递增排序,当把60插入到有序表中时,为寻找其插入位置需比较________次。

答:3。在插入60时,有序区为(4,12,23,48,96),折半查找依次与23、48和96进行比较,比较次数为3。

4. 在对一组记录(50,40,95,20,15,70,60,45,80)进行希尔排序时,第2趟排序结束后前4条记录为________。

答:(15,20,50,40)。第1趟($d_1=4$)后的结果为(15,40,60,20,50,70,95,45,80),第2趟($d_2=2$)后的结果为(15,20,50,40,60,45,80,70,95)。

5. 对数据序列(5,1,7,9,8,6,3,4,2,10)采用冒泡排序方法进行递增排序,每趟通过交换归位关键字最小的元素,经过3趟后的排序结果是________。

答:(1,2,3,5,4,7,9,8,6,10)。

6. 快速排序算法在初始数据________时呈现最差的时间性能。

答:有序。

7. 在记录按关键字有序时快速排序的时间复杂度为________。

答:$O(n^2)$。

8. 有一组关键字序列(50,40,95,20,15,70,60,45,80),采用简单选择排序方法进行递增排序,第4次交换和选择后未排序记录(即无序表)为________。

答:(50,70,60,95,80)。简单选择排序的过程如下:

初始序列:50,40,95,20,15,70,60,45,80
第1趟: 15,40,95,20,50,70,60,45,80
第2趟: 15,20,95,40,50,70,60,45,80
第3趟: 15,20,40,95,50,70,60,45,80
第4趟: 15,20,40,45,50,70,60,95,80

9. 对含有n个元素的数据序列进行简单选择排序,总的关键字比较次数是________。

答:$n(n-1)/2$。简单选择排序中第1趟需要$n-1$次比较,第2趟需要$n-2$次比较,……,第$n-1$趟需要1次比较。

10. 对含有n个元素的数据序列进行简单选择排序,最好情况下元素移动的次数是________。

答:0。当数据序列正序时没有元素移动。

11. 设一组初始关键字为(72,73,71,23,94,16,5),则快速排序的第一趟结果为________。

答:(5,16,71,23,72,94,73)。

12. 堆是一种有用的数据结构。堆排序是一种___①___排序,堆实际上是一棵___②___结点的层次序列。在对含有n个元素的序列进行排序时,堆排序的时间复杂度为___③___,空间复杂度为___④___。关键字序列(5,23,16,68,94,72,71,73)是否满足堆的性质___⑤___。

答:① 选择 ② 完全二叉树 ③ $O(n\log_2 n)$ ④ $O(1)$ ⑤ 满足堆的性质(小根堆)

13. 在堆排序过程中,由n个待排序的记录建立初始堆需要___①___次筛选,由初始堆到排序结束需要进行___②___次筛选。

答：① $\lfloor n/2 \rfloor$ ② $n-1$。

14. 对于关键字序列(12,13,11,18,60,15,7,20,25,100)，用筛选法建堆，应从关键字为________的元素开始。

答：60。将该序列看成一棵完全二叉树，最后一个分支结点是60。

15. 一组关键字序列为(50,40,95,20,15,70,60,45,80)，采用堆排序方法进行递增排序，在建立初始堆后最后4个关键字为________。

答：(50,60,40,20)。建立的初始堆为(95,80,70,45,15,50,60,40,20)。

16. 在二路归并排序中，若待排序记录的个数为20，则共需要进行 ① 趟归并，在第3趟归并中，把长度为 ② 的有序表归并为长度为 ③ 的有序表。

答：①5 ②4 ③8。$n=20$，共需进行$\lceil \log_2 n \rceil=5$趟归并，第1趟(length=1)归并后成为10个有序表，第2趟(length=2)归并后成为5个有序表，第3趟(length=4)归并将长度为4个的有序表归并为长度为8的有序表。

17. 数据序列(8,7,6,5,4,3,2,1)采用二路归并排序方法进行递增排序，所需要的关键字比较次数是________。

答：12。排序过程如下：

8,7,6,5,4,3,2,1 初始数据

7,8,5,6,3,4,1,2 共比较4次

5,6,7,8,1,2,3,4 共比较4次

1,2,3,4,5,6,7,8 共比较4次

18. 在堆排序和快速排序中，若原始记录接近正序或反序，则选用 ① ；若原始记录随机分布，则最好选用 ② 。

答：①堆排序 ②快速排序。

19. 在直接插入和简单选择排序中，若初始数据基本有序，则选用 ① ；若初始数据基本反序，则选用 ② 。

答：①直接插入排序 ②简单选择排序。

20. 在排序过程中，任何情况下都不比较关键字大小的排序方法是________。

答：基数排序。

21. 对于数据序列(288,371,260,531,287,235,56,299,18,23)，采用最低位优先的基数排序进行递增排序，第1趟排序后的结果是________。

答：(260,371,531,023,235,056,287,288,018,299)。

10.3.3 判断题

1. 判断以下叙述的正确性。

(1) 只有在排序数据的初始状态为反序的情况下，冒泡排序过程中元素的移动次数才会达到最大值。

(2) 只有在排序数据的初始状态为反序的情况下，简单选择排序过程中元素的移动次数才会达到最大值。

(3) 对n个元素进行简单选择排序，关键字的比较次数总是$n(n-1)/2$次。

(4) 只有在排序数据的初始状态为反序的情况下，在直接插入排序过程中，元素的移动

次数才会达到最大值。

(5) 只有在排序数据的初始状态为反序的情况下,在堆排序过程中,关键字的比较次数才会达到最大值。

(6) 快速排序和冒泡排序都属于交换类排序方法,每趟产生的有序区都是全局有序的。

答:(1) 正确。

(2) 错误。简单选择排序在初始状态为反序时移动元素的次数会达到最大值,但不仅仅只有这种情况,只要每个元素都不在其最终位置上时都会出现这种情况。

(3) 正确。

(4) 正确。

(5) 错误。

(6) 错误。

2. 判断以下叙述的正确性。

(1) 快速排序方法在任何情况下均可最快得到排序效果。

(2) 基数排序的设计思想是依照对关键字值的比较来实施的。

(3) 排序的稳定性是指排序算法中比较的次数保持不变,且算法能够终止。

(4) 快速排序的速度是所有排序方法中最快的,且所需的辅助空间也最少。

(5) 对一个堆按二叉树层次进行遍历可以得到一个有序序列。

(6) 在任何情况下二路归并排序都比直接插入排序快。

(7) 冒泡排序和快速排序都是基于交换的两种排序方法,前者的最坏时间复杂度为$O(n^2)$,后者的最坏时间复杂度为$O(n\log_2 n)$,所以快速排序在任何情况下都比冒泡排序效率高。

(8) 在任何情况下折半插入排序都优于直接插入排序。

答:(1) 错误。快速排序在待排序记录关键字为随机分布时效果最好,基本有序时效果最差。

(2) 错误。基数排序不进行关键字值的比较。

(3) 错误。

(4) 错误。快速排序的空间复杂度为$O(\log_2 n)$。

(5) 错误。

(6) 错误。二路归并排序最好的时间复杂度为$O(n\log_2 n)$,而直接插入排序最好的时间复杂度为$O(n)$。

(7) 错误。冒泡排序的最好时间复杂度为$O(n)$。

(8) 错误。在排序的数序中元素个数很少时,折半插入排序不一定优于直接插入排序。

3. 判断以下叙述的正确性。

(1) 在执行某个排序算法的过程中出现了元素朝着最终排序序列位置的相反方向移动,则该算法是不稳定的。

(2) 在任何情况下折半插入排序和直接插入排序移动的元素个数一样多。

(3) 希尔排序算法的每一趟都要调用一次或多次直接插入排序算法,所以其效率比直接插入排序算法差。

(4) 冒泡排序在最好情况下元素移动的次数为0。

(5) 如果要从 10 000 个元素中选择前 10 个最小的元素,在二路归并排序和冒泡排序之间选择,应选择二路归并排序。

(6) 快速排序每一趟只能归位无序区中的第一个元素。

(7) 堆排序需要建立初始堆,所以空间复杂度为 $O(n)$。

(8) 一个递增的关键字序列一定构成一个大根堆。

(9) 在大根堆中,堆中任一结点的关键字均大于它的左、右孩子的关键字。

(10) n 个元素采用二路归并排序算法,总的归并趟数为 n。

(11) 基数排序只适用于以数字为关键字的情况,不适用于以字符串为关键字的情况。

(12) 基数排序与初始数据的次序无关。

答:(1) 错误。例如,(12,21)采用基数排序,按个位数排序后为(21,12),而该排序方法是稳定的。

(2) 正确。折半插入排序将直接插入排序中元素的分散移动改为集中移动,移动元素个数不变。

(3) 错误。从平均性能看,希尔排序好于直接插入排序。

(4) 正确。在初始序列正序时冒泡排序中元素移动的次数为 0。

(5) 错误。二路归并排序需要全部排序后才能选择前 10 个最小的元素。

(6) 错误。快速排序每一趟归位一个基准元素,该基准元素可以是无序区中的任何元素。

(7) 错误。

(8) 错误。一个递增的关键字序列一定构成一个小根堆。

(9) 正确。

(10) 错误。二路归并排序的趟数为 $\lceil \log_2 n \rceil$。

(11) 错误。

(12) 正确。

10.3.4 简答题

1. 以关键字序列(15,18,29,12,35,32,27,23,10,20)为例,分别写出执行以下排序算法的各趟排序结束时关键字序列的状态:

(1) 直接插入排序　(2) 希尔排序　(3) 冒泡排序　(4) 快速排序

(5) 简单选择排序　(6) 堆排序　(7) 归并排序

答:(1) 直接插入排序的过程如下。

初始状态:**15**,18,29,12,35,32,27,23,10,20

第 1 趟:　**15,18**,29,12,35,32,27,23,10,20

第 2 趟:　**15,18,29**,12,35,32,27,23,10,20

第 3 趟:　**12,15,18,29**,35,32,27,23,10,20

第 4 趟:　**12,15,18,29,35**,32,27,23,10,20

第 5 趟:　**12,15,18,29,32,35**,27,23,10,20

第 6 趟:　**12,15,18,27,29,32,35**,23,10,20

第 7 趟:　**12,15,18,23,27,29,32,35**,10,20

第 8 趟:　**10,12,15,18,23,27,29,32,35**,20

第 9 趟：　**10,12,15,18,20,23,27,29,32,35**

（2）希尔排序的过程如下。

初始状态：　**15**,18,29,12,35,**32**,27,23,10,20

第 1 趟(d=5)：**15**,18,**23**，10,**20**,32,**27**,29，**12**,35

第 2 趟(d=2)：12,10,15，18,20,29,23,32，27,35

第 3 趟(d=1)：10,12,15，18,20,23,27,29，32,35

（3）冒泡排序的过程如下。

初始状态：15,18,29,12,35,32,27,23,10,20

第 1 趟：　**10**,15,18,29,12,35,32,27,23,20

第 2 趟：　**10,12**,15,18,29,20,35,32,27,23

第 3 趟：　**10,12,15**,18,20,29,23,35,32,27

第 4 趟：　**10,12,15,18**,20,23,29,27,35,32

第 5 趟：　**10,12,15,18,20**,23,27,29,32,35

第 6 趟：　**10,12,15,18,20,23**,27,29,32,35(全部有序)

（4）快速排序的过程如下(每趟给出划分后的结果)。

初始状态：15,18,29,12,35,32,27,23,10,20

第 1 趟：　10,12,**15**,29,35,32,27,23,18,20

第 2 趟：　**10**,12,15,29,35,32,27,23,18,20

第 3 趟：　10,12,15,20,18,23,27,**29**,32,35

第 4 趟：　10,12,15,18,**20**,23,27,29,32,35

第 5 趟：　10,12,15,18,20,**23**,27,29,32,35

第 6 趟：　10,12,15,18,20,23,27,29,**32**,35

（5）简单选择排序的过程如下。

初始状态：15,18,29,12,35,32,27,23,10,20

第 1 趟：　**10**,18,29,12,35,32,27,23,15,20

第 2 趟：　**10,12**,29,18,35,32,27,23,15,20

第 3 趟：　**10,12,15**,18,35,32,27,23,29,20

第 4 趟：　**10,12,15,18**,35,32,27,23,29,20

第 5 趟：　**10,12,15,18,20**,32,27,23,29,35

第 6 趟：　**10,12,15,18,20,23**,27,32,29,35

第 7 趟：　**10,12,15,18,20,23,27**,32,29,35

第 8 趟：　**10,12,15,18,20,23,27,29**,32,35

第 9 趟：　**10,12,15,18,20,23,27,29,32**,35

（6）堆排序(大根堆)的过程如下(每趟给出筛选后的结果)。

初始状态：15,18,29,12,35,32,27,23,10,20

初始堆：　35,23,32,15,20,29,27,12,10,18

第 1 趟：　32,23,29,15,20,18,27,12,10,**35**

第 2 趟：　29,23,27,15,20,18,10,12,**32,35**

第 3 趟：　27,23,18,15,20,12,10,**29,32,35**

第 4 趟：　23,20,18,15,10,12,**27,29,32,35**

第 5 趟：　20,15,18,12,10,**23,27,29,32,35**

第 6 趟：　18,15,10,12,**20,23,27,29,32,35**

第 7 趟：　15,12,10,**18,20,23,27,29,32,35**

第 8 趟：　10,12,**15,18,20,23,27,29,32,35**

第 9 趟：　10,**12,15,18,20,23,27,29,32,35**

(7) 归并排序的过程如下。

初始状态：15,18,29,12,35,32,27,23,10,20

第 1 趟：　15,18,12,29,32,35,23,27,10,20

第 2 趟：　12,15,18,29,23,27,32,35,10,20

第 3 趟：　12,15,18,23,27,29,32,35,10,20

第 4 趟：　10,12,15,18,20,23,27,29,32,35

2. 一组关键字序列为(265,301,751,129,937,863,742,694,076,438),给出以最低位优先的基数排序各趟的排序结果。

答：基数排序的过程如下。

初始状态：　　265,301,751,129,937,863,742,694,076,438

第 1 趟(个位)：301,751,742,863,694,256,076,937,438,129

第 2 趟(十位)：301,129,937,438,742,751,256,863,076,694

第 3 趟(百位)：076,129,256,301,438,694,742,751,863,937

3. 在希尔排序算法(如果初始分组个数 d 为 2 的幂,且每次缩小一半)中,位于每个 d 间隔上的一组数据构成一个数据子序列,然后对这些子序列进行排序。如果这些子序列采用冒泡排序、堆排序、二路归并排序和快速排序,问哪种排序方法比较好？简要说明理由。

答：希尔排序的特点是在开始时 d 较大,分组较多,每组的记录个数少,后来增量 d 逐渐减小,而各组的元素个数逐渐增多。但由于上一次已按各个分组排过序,使数据较接近有序状态。在冒泡排序、堆排序、二路归并排序和快速排序中,只有冒泡排序在数据正序时时间复杂度为 $O(n)$,所以应选择冒泡排序。

4. **证明**：对一个长度为 n 的任意线性表进行排序至少需要进行 $n\log_2 n$ 次比较。

证明：在排序过程中,每次比较(如 a 和 b 两元素进行比较)会出现两条路径(即 $a>b$ 和 $a\leqslant b$ 两种路径),若整个排序过程至少需做 t 次比较,则显然会有 2^t 条路径。由于 n 个记录总共有 $n!$ 种不同的排列,因而必须有 $n!$ 种不同的比较路径,于是有 $2^t\geqslant n!$,即 $t\geqslant \log_2(n!)$。

因为 $\log_2(n!)\approx n\log_2 n$,所以 $t\geqslant n\log_2 n$。

5. 指出堆和二叉排序树的区别。

答：以小根堆为例,堆的特点是双亲结点的关键字必然小于等于孩子结点的关键字,而两个孩子结点的关键字没有次序规定。在二叉排序树中,每个双亲结点的关键字均大于左子树结点的关键字,每个双亲结点的关键字均小于右子树结点的关键字,也就是说,每个双亲结点的左、右孩子的关键字有次序关系。

6. 在对 n 个元素组成的线性表进行快速排序时,所需要进行的比较次数与这 n 个元素的初始排列有关。问：

(1) 当 $n=7$ 时,最好情况下需进行多少次比较?请说明理由。

(2) 当 $n=7$ 时,给出一个最好情况的初始排列的实例。

(3) 当 $n=7$ 时,在最坏情况下需进行多少次比较?请说明理由。

(4) 当 $n=7$ 时,给出一个最坏情况的初始排序的实例。

答:(1) 在最好情况下,假设每次划分能得到两个长度相等的子区间,表的长度 $n=2^k-1$,那么第 1 趟划分得到两个长度为 $\lfloor n/2 \rfloor$ 的子区间,第 2 趟划分得到 4 个长度为 $\lfloor n/4 \rfloor$ 的子区间。依此类推,总共进行 $k=\log_2(n+1)$ 趟划分,各子区间的长度为 1,此时排序完毕。

当 $n=7$ 时,$k=3$,在最好情况下,第 1 趟划分比较 6 次可找到一个其基准是正中间的元素;第 2 趟分别对两个子区间(其长度均为 3,此时 $k=2$)进行划分,各两次比较,得到的都是长度为 1 的子区间,这样就可以将原表排序完毕。所以总共比较 10 次即可,其快速排序判定树与比较次数如图 10.3 所示(图中各结点数字表示对应子区间中的元素个数)。

(2) 当 $n=7$ 时,由(1)可知,每次排序都应使第一个元素归位到表正中位置,因此最好的初始排序情况的例子为(4,7,5,6,3,1,2)。

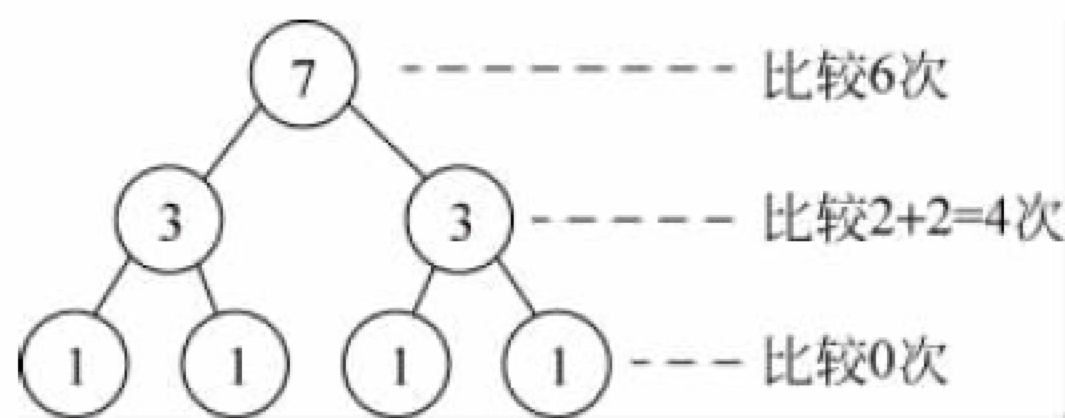


图 10.3 快速排序判定树与比较次数

(3) 在最坏情况下,若每次用来划分的关键字最大(或最小),那么只能得到左(或右)子表,其长度比原长度少 1。因此,若原表中的记录按关键字递减排列,当要求按递增排序时,快速排序的效率与冒泡排序相同,其时间复杂度为 $O(n^2)$,所以当 $n=7$ 时,最坏情况下的比较次数为 21 次。

(4) 当 $n=7$ 时,快速排序在最坏情况下初始排序序列有序,所以 $n=7$ 时,最坏情况的初始排序的例子为(7,6,5,4,3,2,1)。

7. 某关键字序列 R 为(6,2,9,7,3,8,4,5,0,10),用下列各排序方法将 R 中的元素递增排序。

(1) 取第一个元素 6 作为划分基准,给出快速排序第一趟的结果。

(2) 给出将 R 调整成初始堆的过程。

(3) 采用基数为 3 的基数排序法,给出每趟分配和收集后的结果。

答:(1) 快速排序第一趟的结果为(0,2,5,4,3,6,8,7,9,10)。

(2) 将 R 调整成初始堆的过程如图 10.4 所示,所以结果为(10,7,9,6,3,8,4,5,0,2)。

(3) 若以 3 为基数,将关键字序列 R 转换为三进制数,即为(20,2,100,21,10,22,11,12,0,101),最多的位数是 3,补齐后为(020,002,100,021,010,022,011,012,000,101)。

第 1 次分配(以个位数排序)和收集后的结果为:

(020,100,010,000,021,011,101,002,022,012)

第 2 次分配(以个位数排序)和收集后的结果为:

(100,000,101,002,010,011,012,020,021,022)

第 3 次分配(以个位数排序)和收集后的结果为:

(000,002,010,011,012,020,021,022,100,101)

即(0,2,3,4,5,6,7,8,9,10)。

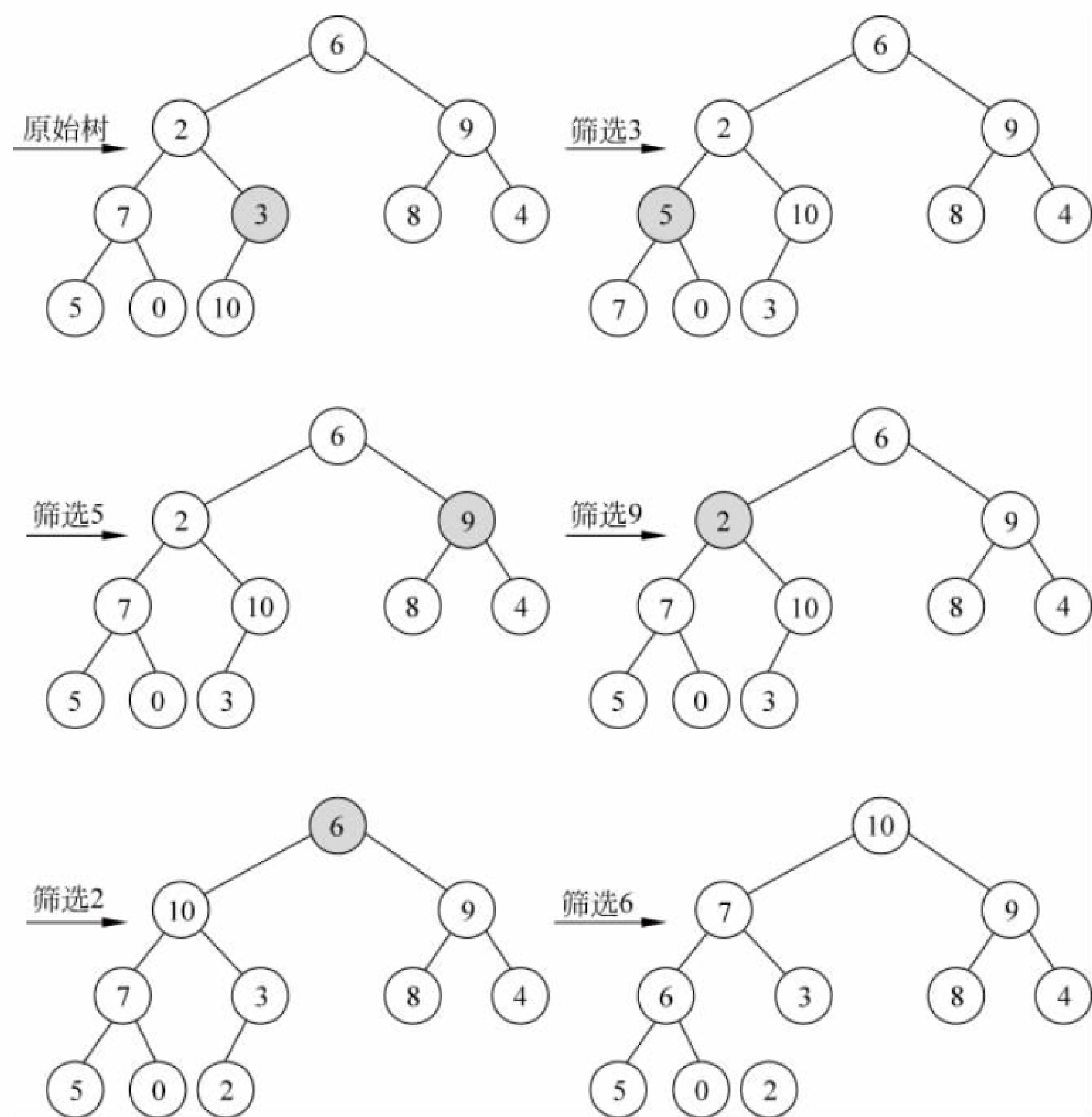


图 10.4 建立初始堆的过程

8. 有以下快速排序算法，指出该算法是否正确，若不正确，请说明错误的原因。

```
void QuickSort(RecType R[ ],int s,int t)        //对 R[s]至 R[t]的元素进行快速排序
{   int i = s,j = t,tmp;
    if (s<t)
    {   tmp = s;
        while (i!= j)
        {   while (j> i && R[j].key > R[tmp].key)
                j--;
            R[i] = R[j];
            while (i<j && R[i].key < R[tmp].key)
                i++;
            R[j] = R[i];
        }
        R[i] = R[tmp];
        QuickSort(R,s,i-1);             //对左区间递归排序
        QuickSort(R,i+1,t);             //对右区间递归排序
    }
}
```

答：不做深入分析，很容易得出该算法正确的结论，其实这个算法是错误的。与正确的快速排序算法进行比较发现，本算法将原来 tmp 保存划分记录的值改为保存划分记录的下标，由于在后面的比较移动中可能改变该 tmp 下标对应的记录值，因而造成执行 $R[i]=R[\text{tmp}]$时基准元素归位错误，从而引起排序失败。

9．在二路归并排序中，对 $R[\text{low..high}]$（$R[\text{low..mid}-1]$和 $R[\text{mid..high}]$是有序的）调用一次二路归并都需要开辟 $O(\text{high}-\text{low}+1)$的辅助空间，共需$\lceil \log_2 n \rceil$趟排序，为什么总的辅助空间仍为 $O(n)$？

答：在二路归并排序中，两个长度为 length 的有序段归并时需要开辟长度为 2length 的辅助空间（即为归并的元素个数），但在一次归并结束后这些辅助空间都被释放了，所以长度为 length 的一趟排序只需要 2length 的辅助空间。而最后一趟排序需要所有记录参与归并，所以总的辅助空间为 $O(n)$。

10．基数排序过程通常用单链表存放排序的元素，是否适合用顺序表来存放排序的元素？为什么？

答：单链表在进行元素插入和删除时不需要移动元素，而基数排序过程涉及大量的元素插入和删除，所以采用单链表存放排序的元素时排序效率更高。如果采用顺序表来存放排序的元素，会大大降低排序性能。

所以说基数排序过程不适合用顺序表来存放排序的元素，但不是说基数排序过程不能够用顺序表来存放排序的元素。

10.3.5　算法设计题

1.【直接插入排序算法】已知顺序表中有 n 个记录，设计一个算法采用直接插入排序方法为该顺序表建立一个有序的索引表（依关键字递增排列），索引表中的每一项应含记录的关键字和该记录在顺序表中的序号。

解：对应的算法如下。

```
typedef struct
{   KeyType key;                                          //关键字
    int no;                                               //序号
}IndexType;                                               //索引类型
void CreateIndex(RecType R[],IndexType idx[],int n)       //建立 R 的索引 idx
{   IndexType tmp;
    int i,j;
    for (i=0;i<n;i++)                                     //建立 idx
    {   idx[i].key=R[i].key;
        idx[i].no=i;
    }
    for (i=1;i<n;i++)          //对 idx 按 key 排序
    {   if (idx[i].key<idx[i-1].key)
        {   tmp=idx[i];
            j=i-1;
            do
            {   idx[j+1]=idx[j];
```

```
                j--;
            } while (j>=0 && idx[j].key>tmp.key);
            idx[j+1]=tmp;
        }
    }
}
```

2.【折半插入排序算法】设计一个折半插入算法将初始数据从大到小递减排序，并要求在初始数据正序时移动元素的次数为零。

解：对应的算法如下。

```
void BinInsertSort3(RecType R[],int n)      //对 R[0..n-1]按递减有序进行折半插入排序
{   int i,j,low,high,mid;
    RecType tmp;
    for (i=1;i<n;i++)
    {   if (R[i-1].key<R[i].key)
        {   tmp=R[i];                       //将 R[i]保存到 tmp 中
            low=0;high=i-1;
            while (low<=high)               //在 R[low..high]中折半查找有序插入的位置
            {   mid=(low+high)/2;           //取中间位置
                if (tmp.key>R[mid].key)
                    high=mid-1;             //插入点在左半区
                else
                    low=mid+1;              //插入点在右半区
            }
            for (j=i-1;j>=high+1;j--)       //元素后移
                R[j+1]=R[j];
            R[high+1]=tmp;                  //插入 R[i]
        }
    }
}
```

3.【冒泡排序算法】设计一个奇偶排序算法，第一趟对所有奇数的 i，将 $R[i]$ 和 $R[i+1]$ 进行比较，第二趟对所有偶数的 i，将 $R[i]$ 和 $R[i+1]$ 进行比较，每次比较时若 $R[i].key>R[i+1].key$，则将两者交换，以后重复上述两趟过程，直到整个数据有序。

解：通过比较 R 中相邻的(奇偶)位置的关键字对，如果该奇偶对是逆序(第一个大于第二个)，则交换。下一步重复该操作，但针对所有的(偶-奇)位置的关键字对，如此交替进行下去。对应的算法如下：

```
void OeSort(RecType R[],int n)
{   int i;
    bool sorted=false;
    while (!sorted)
    {   sorted=true;
        for (i=0;i<n-1;i+=2)              //奇数扫描
            if (R[i].key>R[i+1].key)
            {   sorted=false;
```

```
                swap(R[i],R[i+1]);
            }
        for (i=1;i<n-1;i+=2)                //偶数扫描
            if (R[i].key>R[i+1].key)
            {   sorted=false;
                swap(R[i],R[i+1]);
            }
    }
}
```

4.【快速排序算法】设计快速排序的非递归算法。

解：利用一个含有 low 和 high 两个整数的记录类型的数组 St[]作为栈，其中 low 和 high 分别指示某个子区间的首、尾地址。先将(0，$n-1$)进栈，在栈不空时循环：出栈一个子区间 R[low..high]，对其按 R[low]进行划分，分为两个子区间 R[low..$i-1$]和 R[$i+1$..high]，分别将(low，$i-1$)和($i+1$，high)进栈。对应的算法如下：

```
void QuickSort1(RecType R[],int n)          //对 R[0..n-1]进行快速排序
{   int i,low,high,top=-1;
    struct
    {   int low,high;
    } St[MAXL];
    RecType tmp;
    top++;                                  //进栈
    St[top].low=0; St[top].high=n-1;
    while (top>-1)                          //栈不空取出一个子区间进行划分
    {   low=St[top].low;high=St[top].high;  //出栈
        top--;
        if (low<high)                       //区间内至少存在两个元素的情况
        {   i=partition(R,low,high);        //调用«教程»的划分算法
            top++;                          //左区间进栈
            St[top].low=low;St[top].high=i-1;
            top++;                          //右区间进栈
            St[top].low=i+1;St[top].high=high;
        }
    }
}
```

5.【快速排序算法】在执行快速排序算法时，把栈换为队列对最终的排序结果不会产生任何影响。设计将栈换为队列的非递归快速排序算法。

解：把栈换为队列即可。对应的算法如下：

```
void QuickSort2(RecType R[],int n)              //对 R[0..n-1]进行快速排序
{   int i,low,high;
    int front=-1,rear=-1;                       //队首、队尾指针
    struct
    {   int low;
        int high;
```

```
        } Qu[MaxSize];
        rear++;                                     //进队
        Qu[rear].low = 0;Qu[rear].high = n - 1;
        while (front!= rear)                        //队不空取出一个子区间进行划分
        {   front = (front + 1) % MaxSize;
            low = Qu[front].low;high = Qu[front].high;   //出队
            i = partition(R,low,high);              //调用«教程»的划分算法
            rear = (rear + 1) % MaxSize;
            Qu[rear].low = low;Qu[rear].high = i - 1;    //左区间进队
            rear = (rear + 1) % MaxSize;
            Qu[rear].low = i + 1;Qu[rear].high = high;   //右区间进队
        }
    }
```

6.【简单选择排序算法】采用单链表存放排序的数据，设计相应的简单选择排序算法。

解：用 p 扫描到单链表 L 的数据结点，找到 p 结点开始的最小结点 minp。若 minp 结点不是 p 结点，将它们的值交换。对应的算法如下：

```
void SelSort(LinkNode *&L)
{   LinkNode *p = L->next, *q, *minp;
    ElemType tmp;
    while(p->next!= NULL)               //至少有两个数据结点
    {   minp = p;                       //minp 指向 p 结点开始的最小结点
        q = p->next;
        while (q!= NULL)                //找最小结点 minp
        {   if (q->data < minp->data)
                minp = q;
            q = q->next;
        }
        if (minp!= p)                   //若 minp 结点不是 p 结点,交换两个结点值
        {   tmp = minp->data;
            minp->data = p->data;
            p->data = tmp;
        }
        p = p->next;
    }
}
```

7.【堆排序算法】编写一个算法 HeapInsert(R,k,n)，将关键字 k 插入到大根堆 $R[1..n]$ 中，并保证插入后 R 仍是堆。请分析算法的时间。

解：先将 k 插入 R 中已有元素的尾部（即原堆的长度加 1 的位置，插入后堆的长度加 1），然后从下往上调整，使插入的关键字满足堆性质。对应的算法如下：

```
void HeapInsert(RecType R[],KeyType k,int &n)      //将 k 插入到堆 R[1..n]中
{   int i,j;
    n++;
    R[n].key = k;                                 //增加新值到原表尾部且表长加 1
    i = n/2;j = n;
```

```
    while (i > 0)                       //调整为堆
    {   if (R[i].key < R[j].key)
            swap(R[i],R[j]);            //交换
        j = i;i = i/2;                  //继续自底向上查找
    }
}
```

时间复杂度分析：设该堆对应的树高为 h，则满足 $h \leqslant \log_2 n$，调整是自底向上查找，最多查找到树根，所以时间复杂度为 $O(\log_2 n)$。

8.【堆排序算法】设计一个建堆算法 BuildHeap(R,A,n)，从空堆开始，依次读入元素调用上题中的堆插入算法将其插入堆中。

解：建堆算法如下。

```
void BuildHeap(RecType R[],KeyType A[],int m,int &n)    //建立堆 R[1..n]
{   int i;
    n = 0;                                              //n 为堆中结点个数,初始时为 0
    for (i = 0;i < m;i++)                               //m 为插入的元素个数
        HeapInsert(R,A[i],n);                           //调用上题中的堆插入算法
}
```

9.【堆排序算法】设计一个大根堆删除结点算法 HeapDelete(R,n)，从堆 $R[1..n]$中删去 $R[1]$并调整为堆。

解：将 $R[1]$与 $R[n]$交换，结点个数减 1，再筛选为堆。堆删除的算法如下：

```
void HeapDelete(RecType R[],int &n)             //将 R[1]从 R[1..n]的堆中删除
{   swap(R[1],R[n]);
    n--;
    sift(R,1,n);                                //调用«教程»的筛选算法
}
```

10.【堆排序算法】设计一个算法，判断一个数据序列 $R[1..n]$是否构成一个大根堆。

解：当元素个数 n 为偶数时，最后一个分支结点（编号为 $n/2$）只有左孩子（编号为 n），其余分支结点均为双分支结点；当 n 为奇数时，所有分支结点均为双分支结点。对每个分支结点进行判断，只有一个分支结点不满足大根堆的定义，返回 false；如果所有分支结点均满足大根堆的定义，返回 true。对应的算法如下：

```
bool IsHeap(RecType R[],int n)
{   int i;
    if (n % 2 == 0)                         //n 为偶数时
    {   if (R[n/2].key < R[n].key)          //最后一个分支结点只有左孩子(编号为 n)
            return false;
        for (i = n/2 - 1;i >= 1;i--)        //判断所有双分支结点
            if (R[i].key < R[2 * i].key || R[i].key < R[2 * i + 1].key)
                return false;
    }
```

```
        else                                                //n为奇数时
        {    for (i = n/2;i >= 1;i -- )                     //所有分支结点均为双分支结点
                if (R[i].key < R[2 * i].key || R[i].key < R[2 * i + 1].key)
                    return false;
        }
        return true;
    }
```

11.【计数排序算法】有一种简单的排序算法叫计数排序，这种排序算法对一个待排序的表进行排序，并将排序结果存放到另一个新的表中。必须注意的是，表中所有待排序的关键字互不相同。计数排序算法针对表中的每个记录，扫描待排序的表一趟，统计表中有多少个记录的关键字比该记录的关键字小。假设针对某一个记录，统计出的计数值为 count，那么这个记录在新的有序表中的合适存放位置即为 count。

(1) 设计计数排序的算法。

(2) 对于有 n 个记录的表，关键字的比较次数是多少？

(3) 与简单选择排序相比较，这种方法是否更好？为什么？

解：(1) 计数排序的算法如下(由无序表 $A[0..n-1]$产生有序表 $B[0..n-1]$)。

```
void CountSort(RecType A[],RecType B[],int n)
{    int i,j,count;
     for (i = 0;i < n;i++)
     {    count = 0;
          for (j = 0;j < n;j++)
               if (A[j].key < A[i].key)             //统计小于 A[i].key 的记录个数
                    count++;
          B[count] = A[i];                          //A[i]应为第 count 大的记录
     }
}
```

(2) 对于有 n 个记录的表，i 从 $0\sim n-1$ 循环，j 从 $0\sim n-1$ 循环，每次循环进行一次关键字比较，所以关键字的比较次数为 n^2。

(3) 简单选择排序比计数排序更好，因为对具有 n 个记录的数据表进行简单选择排序只需进行 $n(n-1)/2$ 次比较，且可原地进行排序。

第11章 外排序

11.1 本章知识体系



1. 知识结构图

本章的知识结构如图 11.1 所示。

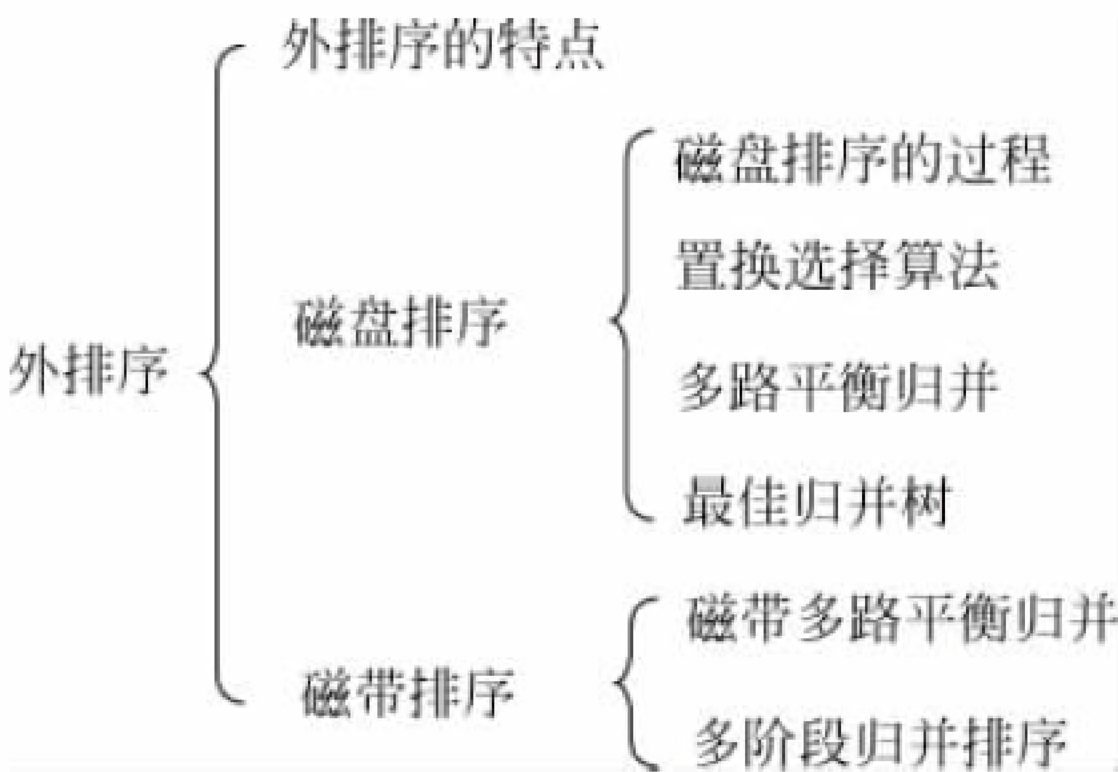


图 11.1 第 11 章知识结构图

2. 基本知识点

(1) 外排序的基本步骤。

(2) 败者树的构造过程及在磁盘排序中的应用。

(3) 最佳归并树的构造过程。

(4) 磁带多路平衡归并过程。

3. 要点归纳

(1) 外排序是指排序过程中需要进行内、外存数据交换的排序方法。

(2) 外排序的过程是先生成若干初始归并段,然后进行多路归并。

(3) 外排序的主要时间花费在关键字的比较和记录的读写上,可以简单理解为外排序执行时间=关键字比较时间+记录读写时间。

(4) 置换-选择排序算法用于生成初始归并段,通常产生的初始归并段个数较少。

(5) 在多路平衡归并中,归并路数 k 越大,记录的读写次数会越小。

(6) 在多路平衡归并中采用简单比较时,k 越大,关键字的比较次数会越大。

(7) 在多路平衡归并中采用败者树时,关键字的比较次数与 k 无关,所以 k 越大越好。

(8) 败者树用于在归并中从 k 个记录中挑选最小记录。

(9) 最佳归并树给出了一种记录读写最少的归并方案。

11.2 教材中的练习题及参考答案



1. 外排序中两个相对独立的阶段是什么?

答:外排序中两个相对独立的阶段是产生初始归并段和多路归并排序。

2. 给出一组关键字 T=(12,2,16,30,8,28,4,10,20,6,18),设内存工作区可容纳 4 个记录,给出用置换-选择排序算法得到的全部初始归并段。

答:置换-选择排序算法的执行过程如表 11.1 所示,共产生两个初始归并段,归并段 1 为(2,8,12,16,28,30),归并段 2 为(4,6,10,18,20)。

表 11.1 初始归并段的生成过程

读入记录	内存工作区状态	R_{min}	输出之后的初始归并段状态
12,2,16,30	12,2,16,30	2(i=1)	归并段 1:{2}
8	12,8,16,30	8(i=1)	归并段 1:{2,8}
28	12,28,16,30	12(i=1)	归并段 1:{2,8,12}
4	4,28,16,30	16(i=1)	归并段 1:{2,8,12,16}
10	4,28,10,30	28(i=1)	归并段 1:{2,8,12,16,28}
20	4,20,10,30	30(i=1)	归并段 1:{2,8,12,16,28,30}
6	4,20,10,6	4(4<30,开始新归并段 i=2)	归并段 1:{2,8,12,16,28,30} 归并段 2:{4}
18	18,20,10,6	6(i=2)	归并段 1:{2,8,12,16,28,30} 归并段 2:{4,6}
	18,20,10,	10(i=2)	归并段 1:{2,8,12,16,28,30} 归并段 2:{4,6,10}
	18,20,,	18(i=2)	归并段 1:{2,8,12,16,28,30} 归并段 2:{4,6,10,18}
	,20,,	20(i=2)	归并段 1:{2,8,12,16,28,30} 归并段 2:{4,6,10,18,20}

3. 设输入的关键字满足 $k_1>k_2>\cdots>k_n$,缓冲区大小为 m,用置换-选择排序方法可产生多少个初始归并段?

答:可产生$\lceil n/m \rceil$个初始归并段。设记录 R_i 的关键字为 $k_i(1\leqslant i\leqslant n)$,先读入 m 个记录 R_1、R_2、…、R_m,采用败者树选择最小记录 R_m,将其输出到归并段 1,$R_{min}=k_m$,在该位置上读入 R_{m+1},采用败者树选择最小记录 R_{m-1},将其输出到归并段 1,$R_{min}=k_{m-1}$,在该位置上读入 R_{m+2},采用败者树选择最小记录 R_{m-2},将其输出到归并段 1,$R_{min}=k_{m-2}$,…,以此类推,产生归并段 1:($R_m,R_{m-1},\cdots,R_1$)。同样产生其他归并段($R_{2m},R_{2m-1},\cdots,R_{m+1}$),($R_{3m},R_{3m-1},\cdots,R_{2m+1}$),…,一共有$\lceil n/m \rceil$个初始归并段。

4. 什么是多路平衡归并?多路平衡归并的目的是什么?

答:归并过程可以用一棵归并树来表示。在多路平衡归并对应的归并树中,每个结点都是平衡的,即每个结点的所有子树的高度相差不超过 1。

k 路平衡归并的过程是第一趟归并将 m 个初始归并段归并为$\lceil m/k \rceil$个归并段,以后每一趟归并将 l 个初始归并段归并为$\lceil l/k \rceil$个归并段,直到最后形成一个大的归并段为止。

m 个归并段采用 k 路平衡归并,总的归并趟数 $s=\lceil \log_k m \rceil$。其趟数是所有归并方案中最少的,所以多路平衡归并的目的是减少归并趟数。

5. 什么是败者树?其主要作用是什么?用于 k 路归并的败者树中共有多少个结点(不

计冠军结点)?

答:败者树是一棵有 k 个叶子结点的完全二叉树,从叶子结点开始,两个结点进行比较,将它们中的败者(较大者)上升到双亲结点,胜者(较小者)参加更高一层的比较。

败者树的主要作用是从 k 个记录中选取关键字最小的记录。

败者树中有 k 个叶子结点,且没有度为 1 的结点,即 $n_0=k$,$n_1=0$,$n_2=n_0-1=k-1$,所以 $n=n_0+n_1+n_2=2k-1$。

6. 如果某个文件经内排序得到 80 个初始归并段,试问:

(1) 若使用多路平衡归并执行 3 趟完成排序,那么应取的归并路数至少应为多少?

(2) 如果操作系统要求一个程序同时可用的输入/输出文件的总数不超过 15 个,则按多路平衡归并至少需要几趟可以完成排序?如果限定这个趟数,可取的最低路数是多少?

答:(1) 设归并路数为 k,初始归并段个数 $m=80$,根据多路平衡归并趟数计算公式 $s=\lceil \log_k m \rceil=\lceil \log_k 80 \rceil=3$,则 $k^3\geqslant 80$,即 $k\geqslant 5$。也就是说,可取的最低路数是 5。

(2) 设多路平衡归并的归并路数为 k,需要 k 个输入缓冲区和一个输出缓冲区。一个缓冲区对应一个文件,有 $k+1=15$,因此 $k=14$,可做 14 路归并。由 $s=\lceil \log_k m \rceil=\lceil \log_{14} 80 \rceil=2$,即至少需两趟归并可完成排序。

若限定这个趟数,由 $s=\lceil \log_k 80 \rceil=2$,有 $80\geqslant k^2$,可取的最低路数为 9,即要在两趟内完成排序,进行 9 路排序即可。

7. 若采用置换选择排序算法得到 8 个初始归并段,它们的记录个数分别为 37、34、300、41、70、120、35 和 43。画出这些磁盘文件进行归并的 4 阶最佳归并树,计算出总的读写记录数。

答:$k=4$,$m=8$,$k-(m-1) \bmod (k-1)-1=2$,设两个虚段。4 阶最佳归并树如图 11.2 所示。

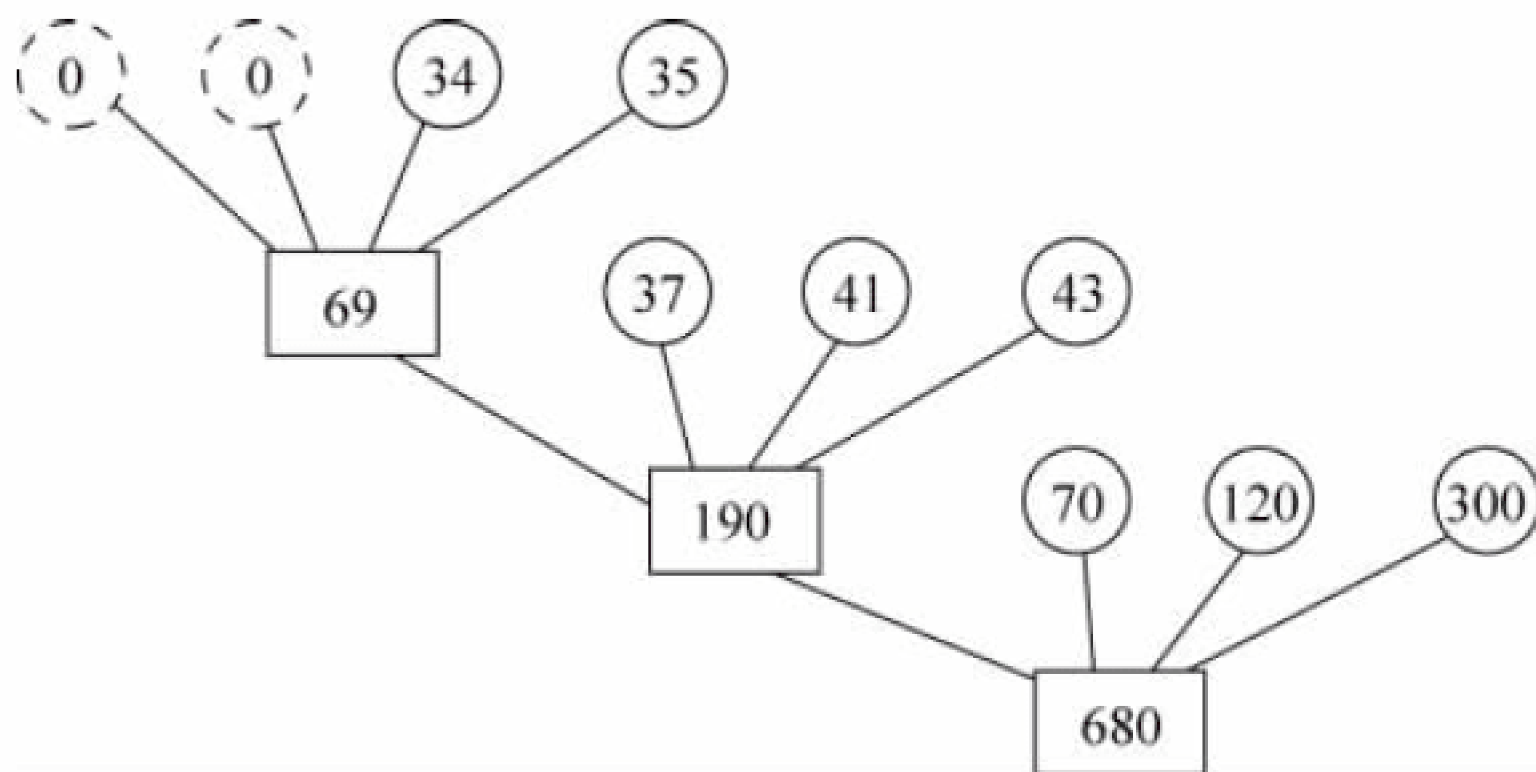


图 11.2 一棵 4 阶最佳归并树

第 1 趟读记录数:34+35=69。

第 2 趟读记录数:69+37+41+43=190。

第 3 趟读记录数:190+70+120+300=680。

总的读记录数=69+190+680=939,总的读写记录数=939×2=1878。

11.3 补充练习题及参考答案



11.3.1 单项选择题

1. 外排序和内排序的主要区别是________。

A. 内排序速度快，而外排序速度慢

B. 内排序不涉及内、外存数据交换，而外排序涉及内、外存数据交换

C. 内排序所需的内存小，而外排序所需的内存大

D. 内排序的数据量小，而外排序的数据量大

答：B。

2. 多路平衡归并的目的是________。

A. 减少初始归前段的段数　　B. 便于实现败者树

C. 减少归并趟数　　D. 以上都对

答：C。

3. 当内存工作区可容纳的记录个数 $w=2$ 时，记录序列(5,4,3,2,1)采用置换-选择算法产生________个递增有序段。

A. 1　　B. 2　　C. 3　　D. 5

答：C。

4. 有 m 个初始归并段，在采用 k 路归并时，所需的归并遍数是________。

A. $\log_2 k$　　B. $\log_2 m$　　C. $\log_k m$　　D. $\lceil \log_k m \rceil$

答：D。

5. 设有 100 个初始归并段，如采用 k 路平衡归并 3 遍完成排序，则 k 值最大是________。

A. 3　　B. 4　　C. 5　　D. 6

答：$m=100$，$\lceil \log_k m \rceil=3$，k 的最大值为 5。本题的答案为 C。

6. m 个初始归并段采用 k 路平衡归并时，构建的败者树中共有________个结点(不计冠军结点)。

A. $2k-1$　　B. $2k$　　C. $2m$　　D. $2m-1$

答：A。

7. 在用败者树进行 k 路平衡归并的外排序算法中，总的关键字比较次数与 k ________。

A. 成正比　　B. 成反比　　C. 无关　　D. 以上都不对

答：C。

8. 对于磁带文件的 k 路多阶段归并需要使用________台磁带机。

A. 1　　B. $2k$　　C. k　　D. $k+1$

答：D。

11.3.2 填空题

1. 归并外排序有两个基本阶段，第一个阶段是__①__，第二个阶段是__②__。

答：①生成初始归并段 ②多路归并。

2. 外排序的基本方法是归并排序，但在之前必须先生成________。

答：初始归并段。

3. 置换-选择排序算法的作用是________。

答：由一个无序文件产生若干个有序子文件(初始归并段)。

4. 一组关键字(12,2,16,30,8,28,4,10,20,6,18)，设内存工作区可容纳4个记录，采用置换-选择排序，则产生________个初始归并段。

答：2。产生的初始归并段为(2,8,12,16,28,30)和(4,6,10,18,20)。

5. m 个初始归并段采用 k 路平衡归并，归并的趟数是________。

答：$\lceil \log_k m \rceil$。

6. 在选择树中，"败者"是指________。

答：在一次比较中没有上升而进入其双亲的结点(递增排序中为较大者)。

7. 对于98个长度不等的初始归并段，在构建5路最佳归并树时需要增加________个虚段。

答：3。$k=5, m=98, k-(m-1)\bmod(k-1)-1=3$。

8. 对于磁带文件的 k 路平衡归并最好使用________台磁带机。

答：$2k$。

11.3.3 判断题

1. 判断以下叙述的正确性。

(1) 外排序与外部设备的特性无关。

(2) 外排序是把外存文件调入内存，可以利用内排序的方法进行排序，因此排序所花的时间只取决于内排序的时间。

(3) 在外排序的 k 路平衡归并中，当采用败者树时，关键字的比较次数与 k 无关。

(4) 采用多路平衡归并方法可以减少初始归并段的个数。

(5) 在对磁盘文件进行 k 路平衡归并排序时，k 值越大，所需的归并趟数越少。

答：(1) 错误。

(2) 错误。外排序的时间还包括记录读写的时间。

(3) 正确。

(4) 错误。多路平衡归并可以减少归并的趟数。

(5) 正确。归并趟数为$\lceil \log_k m \rceil$。

2. 判断以下叙述的正确性。

(1) 内排序过程在数据量很大时就变成了外排序过程。

(2) k 路平衡归并建立的败者树中没有度为1的结点。

(3) 置换-选择排序算法的作用是由一个无序文件生成若干个有序的子文件。

(4) k 路最佳归并树在外排序中的作用是设计 k 路归并排序的优化方案。

(5) k 路最佳归并树在外排序中的作用是产生初始归并段。

(6) 对于磁带文件的 k 路平衡归并最好使用 $k+1$ 台磁带机。

答:(1) 错误。

(2) 正确。

(3) 正确。

(4) 正确。

(5) 错误。

(6) 错误。k 路平衡归并最好使用 $2k$ 台磁带机,k 台作为输入,k 台作为输出。

11.3.4 简答题

1. 设输入的关键字满足 $k_1<k_2<\cdots<k_n$,缓冲区大小为 m,用置换-选择排序方法可产生多少个初始归并段?

答:可产生一个初始归并段。设记录 R_i 的关键字为 $k_i(1\leqslant i\leqslant n)$,先读入 m 个记录 $R_1,R_2,\cdots,R_m$,采用败者树选择最小记录 R_1,将其输出到归并段 1,$R_{\min}=k_1$,在该位置上读入 R_{m+1},采用败者树选择最小记录 R_2,由于 $k_2>R_{\min}$,将其输出到归并段 1,$R_{\min}=k_2$,在该位置上读入 R_{m+2},……,如此,只产生一个初始归并段,其记录为$(R_1,R_2,\cdots,R_n)$。实际上,由于关键字满足 $k_1<k_2<\cdots<k_n$,它已是一个初始归并段了。

2. 文件中记录的关键字序列为(41,39,28,32,22,19,11,50,13,21,1,33,37,3,52,16,4,8,72,12,32),设缓冲区 WA 有容纳 5 个记录的容量。按置换-选择排序方法求初始归并段。

答:采用置换-选择排序方法求初始归并段如下。

归并段 1:22,28,32,39,41,50。

归并段 2:1,11,13,19,21,33,37,52,72。

归并段 3,4,8,12,16,32。

3. 以归并排序为例说明内排序和外排序的不同,说明外排序如何提高操作效率?

答:内排序中的归并排序是在内存中进行的归并排序,所有数据都要调入内存,所需的辅助空间为 $O(n)$。外排序的归并排序是将外存中的多个有序子文件合并成一个有序子文件。

外排序的效率主要取决于读写外存的次数,当 m 个初始子文件采用 k 路平衡归并时,其归并趟数 $s=\lceil\log_k m\rceil$,s 越大读写外存的次数也越大,增大 k 和减少 m 都可以减少 s,从而提高外排序的效率。

4. 外排序中的“败者树”和堆有什么区别?若用败者树求 k 个数中的最小值,在某次比较中得到 $a>b$,那么谁是败者?

答:外排序中的“败者树”和堆的区别如下。

败者树是在双亲结点中记下刚进行比较的败者(较大者),让胜者(较小者)去参加更高一层的比较。而堆可看作是一种“胜者树”,即双亲结点表示其左、右孩子中的胜者。

败者树中参加比较的 k 个关键字全部为叶子结点,双亲结点即为左、右孩子的败者,败者树中的结点总数为 $2k-1$,加上冠军结点,总结点个数为 $2k$。堆是由 k 个关键字组成的完全二叉树,每个关键字作为树中的一个结点,根结点是 k 个关键字中的胜者,树中结点总数

为 k。

若用败者树求 k 个数中的最小者，在某次比较中得到 $a>b$，那么 a 是败者。

5. 简述最佳归并树的作用。

答：在多路归并中，如果各初始归并段的长度不同，对它们进行读写的记录次数也不同，通过构造相应的最佳归并树（即 k 叉哈夫曼树）产生最优的归并方案，按照这个方案执行会产生最小的记录读写次数。

6. 有 m 个初始归并段，在构建 k 路最佳归并树时为什么在有些情况下要增加若干个虚段？

答：k 路最佳归并树是一棵 k 阶哈夫曼树，其中只有度为 0 和度为 k 的结点。

设树中结点个数为 n，$n_0=m$，度之和 $=n-1=kn_k$，$n=n_0+n_k$，所以有 $kn_k=m+n_k-1$，则 $n_k=(m-1)/(k-1)$。显然 n_k 一定为正整数，如果 m、k 的初值不当，会导致 n_k 不为正整数，此时会出现少于 k 个的归并段进行 k 路归并，这是十分麻烦的。

为了保证每次都是 k 个归并段进行归并，则 $n_k=(m-1)/(k-1)$ 应为正整数，或者说 $u=(m-1)\ \%\ (k-1)=0$，若 $u\neq0$（不妨设 $m-1=x(k-1)+u$，其中 x 为整数），需添加若干个长度为 0 的虚段，显然添加最少的虚段个数为 $k-1-u$。这样添加后初始归并段为 $m+k-1-u$ 个，新的 $n_k=(m+k-1-u-1)/(k-1)=[x(k-1)+u+k-u-1]/(k-1)=x+1$ 为正整数。

7. 设有 11 个长度不同的初始归并段，它们所包含的记录个数分别为(25,40,16,38,77,64,53,88,9,48,98)。试根据它们做 4 路平衡归并，要求：

(1) 指出总的归并趟数；

(2) 构造最佳归并树；

(3) 根据最佳归并树计算每一趟及总的读记录数。

答：(1) 总的归并趟数 $=\lceil\log_4 11\rceil=2$。

(2) $n=11$，$k=4$，$(n-1)\%(k-1)=1\neq0$，需要附加 $k-1-(n-1)\%(k-1)=2$ 个长度为 0 的虚归并段，最佳归并树如图 11.3 所示。

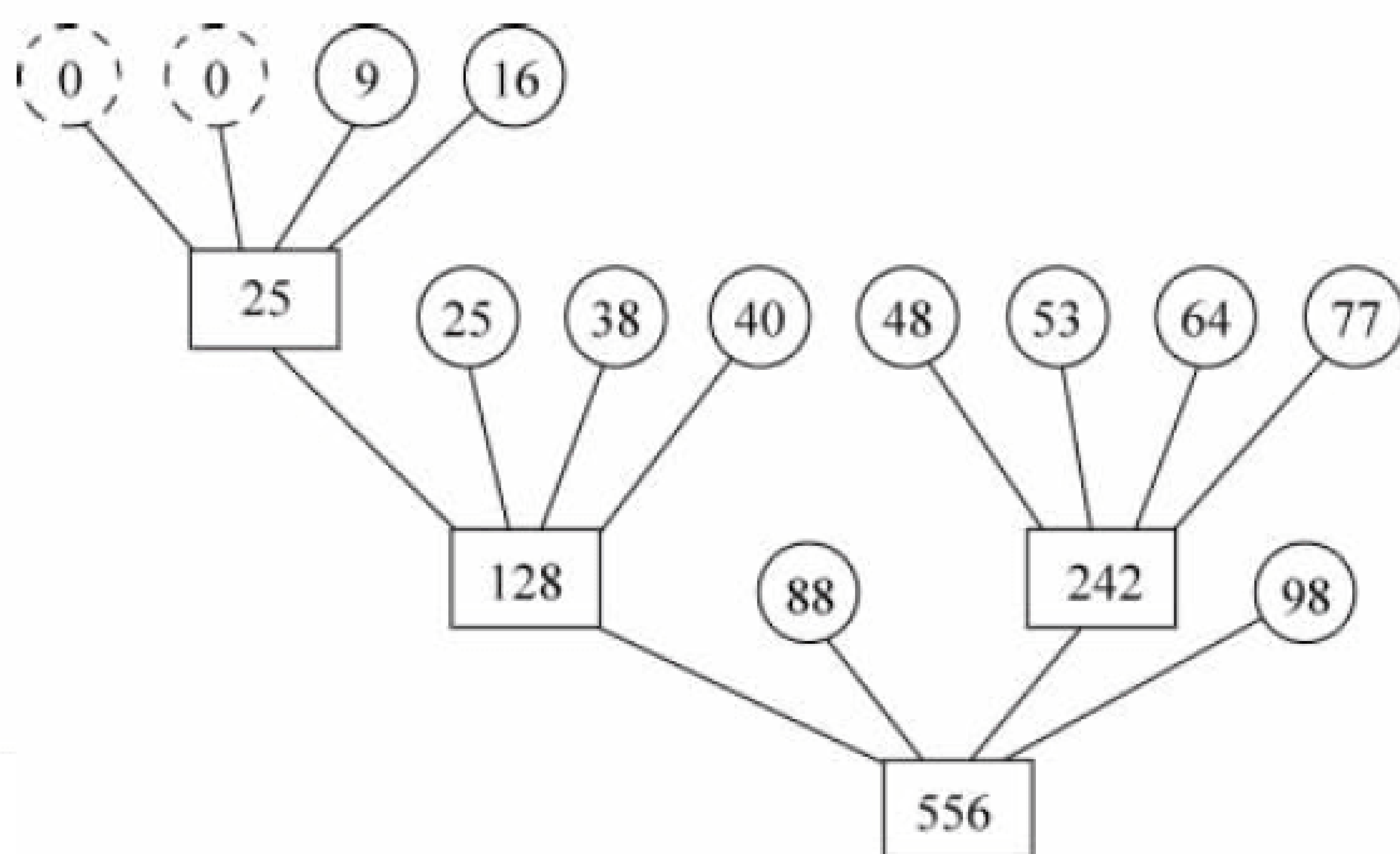


图 11.3 一棵 4 阶最佳归并树

(3) 根据最佳归并树计算每一趟及总的读记录次数：

第 1 趟的读记录数 $=9+16=25$。

第 2 趟的读记录数 $=25+25+38+40+48+53+64+77=370$。

第 3 趟的读记录数＝128＋88＋242＋98＝556。

总的读记录数＝25＋370＋556＝951。

8. 设有 13 个初始归并段，长度分别为(28，16，37，42，5，9，13，14，20，17，30，12，18)。试画出 4 路归并时的最佳归并树，并计算它的带权路径长度 WPL。

答：$n=13$，$k=4$。$(n-1)\%(k-1)=0$，不需加虚段。最佳归并树如图 11.4 所示。

WPL＝(5＋9＋12＋13＋14＋16＋17＋18＋20＋28＋30＋37)×2＋42＝480。

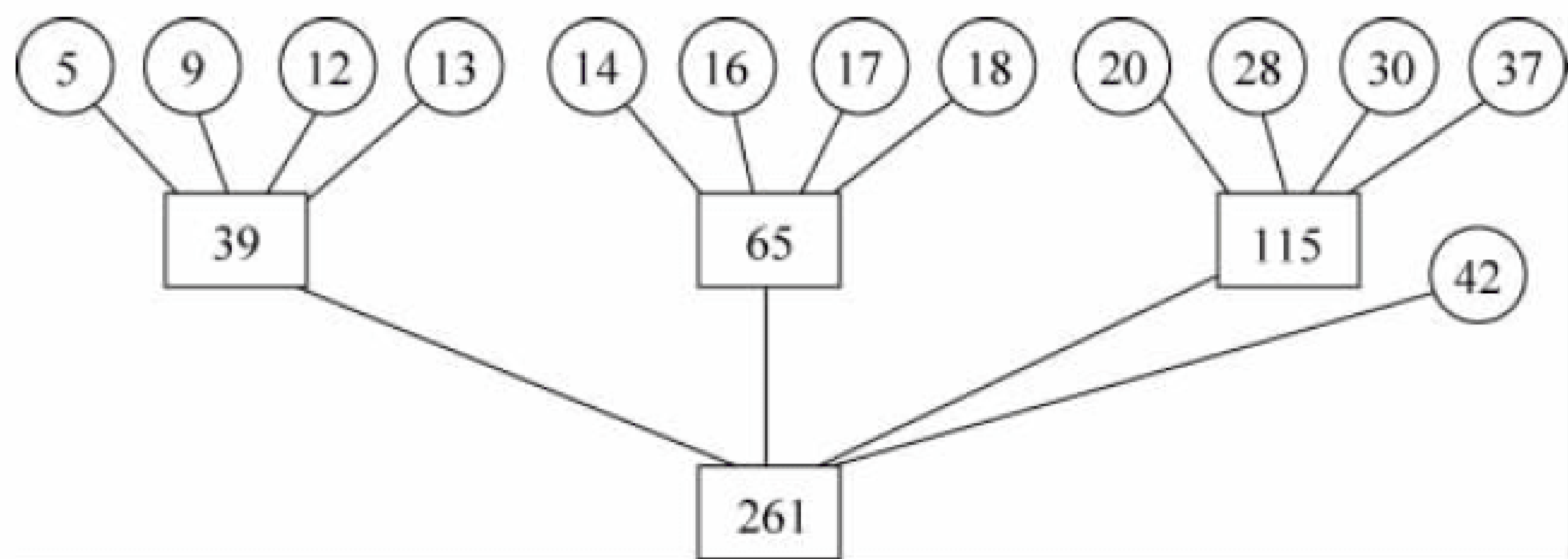


图 11.4　一棵 4 阶最佳归并树

9. 有 55 个长度为 L 的归并段，用 2 路多阶段归并法进行排序，写出归并过程中各磁带内容的变化情况。

答：$k=2$，2 阶 Fibonacci 序列为 0，1，1，2，3，5，8，13，21，44，65，…。$T=2\times22+13=55$，$i=8$，所以初始分布为：

$T_1=21+13=34$，$T_2=21$，$T_3=0$，归并阶段数＝$i-k+2=8$。

其各阶段中归并段的段数在各磁带机上的分布情况如表 11.2 所示。

表 11.2　三带二路归并各阶段中归并段的段数分布情况

阶段号	T_1	T_2	T_3	归并段总数
初始	34	21	0	55
1	13	0	21	34
2	0	13	8	21
3	8	5	0	13
4	3	0	5	8
5	0	3	2	5
6	2	1	0	3
7	1	0	1	2
8	0	1	0	1

10. 用 6 台磁带进行 5 路合并，有 497 个长度相等的归并段，设计初始分布，使排序后的文件在 T_1 上。

答：$k=5$，5 阶 Fibonacci 序列为 0，0，0，0，1，1，2，4，8，16，31，61，120，236，464，912，…。$T=5\times61+4\times31+3\times16+2\times8+4=497$，即 $i=11$，所以初始分布为：

$T_1=61$，$T_2=61+31=92$，$T_3=61+31+16=108$，$T_4=61+31+16+8=116$，$T_5=61+31+16+8+4=120$，$T_6=0$。

归并阶段数＝$i-k+2=8$。其各阶段中归并段的段数在各磁带机上的分布情况如

表 11.3 所示。

表 11.3　六带五路归并各阶段中归并段的段数分布情况

阶段号	T_1	T_2	T_3	T_4	T_5	T_6
初始	61	92	108	116	120	0
1	0	31	47	55	59	61
2	31	0	16	24	28	30
3	15	16	0	8	12	14
4	7	8	8	0	4	6
5	3	4	4	4	0	2
6	1	2	2	2	2	0
7	0	1	1	1	1	1
8	1	0	0	0	0	0

第12章 文件

12.1 本章知识体系



1. 知识结构图

本章的知识结构如图 12.1 所示。



图 12.1　第 12 章知识结构图

2. 基本知识点

（1）文件的逻辑结构和存储结构。

（2）索引文件基本结构、ISAM 和 VSAM 两类索引文件结构。

（3）哈希文件结构、哈希文件和哈希表的异同。

（4）多关键字文件结构。

3. 要点归纳

（1）文件是性质相同的记录的集合。文件的数据量通常很大，它被放置在外存上。

（2）文件的主关键字是唯一标识一个记录的数据项或数据项的组合。次关键字不一定能够唯一标识一个记录。

（3）文件结构包括逻辑结构、存储结构以及在文件上的各种操作（运算）。

（4）顺序文件按记录进入文件的先后顺序存放，其逻辑顺序跟物理顺序一致。

（5）索引文件是由索引表和主文件一起构成的。ISAM 文件和 VSAM 文件都属于索引文件。

（6）哈希文件利用哈希存储方式组织文件，包含哈希函数和处理冲突的方法。

（7）多关键字文件包含有多个次关键字索引，多重表文件和倒排文件都属于多关键字文件。

12.2 教材中的练习题及参考答案



1. 什么是文件的逻辑记录和物理记录？它们有什么区别与联系？

答：记录是文件存取操作的基本单位。逻辑记录是按用户观点描述的基本存取单位，物理记录是按外存设备观点描述的基本存取单位。

通常逻辑记录和物理记录之间存在下面 3 种关系：

（1）一个物理记录存放一个逻辑记录。

（2）一个物理记录包含多个逻辑记录。

（3）多个物理记录表示一个逻辑记录。

2. 比较顺序文件、索引非顺序文件、索引顺序文件和哈希文件的存储代价、检索、插入及删除记录时的优点和缺点。

答：（1）顺序文件只能按顺序查找法存取，即按记录的主关键字逐个查找。这种查找法对于少量的检索是不经济的，但适合于批量检索。顺序文件的存取优点是速度快。顺序文件不能按像顺序表那样的方法进行插入、删除和修改，因为文件中的记录不能像数组空间的数据那样“移动”，而只能通过复制整个文件的方法来实现上述更新操作。

（2）索引非顺序文件适合于随机存取，这是由于主文件的记录是未按关键字排序的，若要进行顺序存取将会频繁地引起磁头移动，因此索引非顺序文件不适合于顺序存取。在索引顺序文件中，由于主文件也是有序的，所以它既适合于直接存取，也适合于顺序存取。另一方面，索引非顺序文件的索引是稠密索引，而索引顺序文件是稀疏索引，故后者的索引减少了索引项的数目，虽然它不能进行“预查找”，但由于索引占用的空间较少、管理要求低，因而提高了索引表的查找速度。

（3）哈希文件也称为直接存取文件，是利用哈希法组织文件，它类似于哈希表，根据文件中关键字的哈希函数值和处理冲突的方法将记录哈希到外存上。这种文件组织方法只适用于像磁盘这样的直接存取设备。哈希文件的优点是文件可以随机存放，记录不必进行排序，插入、删除操作方便，存取速度快，无须索引区，因而节省存储空间。哈希文件的缺点是不能进行顺序存取且访问方式也只限于简单查询，此外在经过多次插入、删除后可能出现文件结构不合理以及记录分布不均匀等现象，这时需要重组文件，但这个工作是很费时的。

3. 某职工表文件如表12.1所示，其中以职工号为主关键字。

表12.1 职工表

物理地址	职工号	姓名	年龄	性别	工作时间	职称
1	105	李华	36	男	1997.7	副教授
2	125	王丽	42	女	1984.7	副教授
3	108	张英	28	男	1998.7	讲师
4	182	陈军	52	男	1970.9	教授
5	135	吴斌	25	男	1999.9	助教
6	116	章萍	58	女	1965.9	教授
7	140	华明	40	男	1989.7	讲师

（1）若将职工文件组织成顺序有序文件，请给出文件的存储结构。

（2）若将该文件组织成索引非顺序文件，请给出其索引表结构。

（3）若将该文件组织成多重表文件，请给出主文件结构及性别索引，工作时间索引（只考虑年份）及年龄段（10岁为一个年龄段）索引。

（4）若将该文件组织成倒排文件，请写出性别索引、职称索引及年龄段索引，组织索引要求同（3）。

答：（1）该文件若组织为顺序有序文件，则应将职工文件记录按职工号有序存放，每个记录信息不变。假设其顺序有序文件仍存储在原文件区域，则它的文件结构如表12.2所示

(假设物理地址编号从 1 开始)。

表 12.2 职工的顺序有序文件存储结构

物理地址	职工号	姓名	年龄	性别	工作时间	职称
1	105	李华	36	男	1997.7	副教授
2	108	张英	28	男	1998.7	讲师
3	116	章萍	58	女	1965.9	教授
4	125	王丽	42	女	1984.7	副教授
5	135	吴斌	25	男	1999.9	助教
6	140	华明	40	男	1989.7	讲师
7	182	陈军	52	男	1970.9	教授

(2) 若该文件组织成索引非顺序文件,则主文件即为表 12.1 所示的职工文件,其索引文件如表 12.3 所示(假设物理地址编号从 11 开始)。

表 12.3 索引表

物理地址	职工号	物理地址
11	105	1
12	108	3
13	116	6
14	125	2
15	135	5
16	140	7
17	182	4

(3) 若将该文件组织成多重链表文件,其主文件如表 12.4 所示,其性别索引、职称索引和年龄索引分别如表 12.5、表 12.6 和表 12.7 所示。

表 12.4 多重表主文件

物理地址	职工号	姓名	年龄	性别	工作时间	职称	性别链	年龄链	职称链
1	105	李华	36	男	1997.7	副教授	3	∧	2
2	125	王丽	42	女	1984.7	副教授	6	7	∧
3	108	张英	28	男	1998.7	讲师	4	5	7
4	182	陈军	52	男	1970.9	教授	5	6	6
5	135	吴斌	25	男	1999.9	助教	7	∧	∧
6	116	章萍	58	女	1965.9	教授	∧	∧	∧
7	140	华明	40	男	1989.7	讲师	∧	∧	∧

表 12.5 性别索引

次关键字	头指针	链长
男	1	5
女	2	2

表 12.6 职称索引

次关键字	头指针	链长
助教	5	1
讲师	3	2
副教授	1	2
教授	4	2

表 12.7 年龄索引

次关键字	头指针	链长
20-29	3	2
30-39	1	1
40-49	2	2
50-59	4	2

(4) 若将该文件组织成倒排文件,主文件如表 12.1 所示,对应的性别索引、职称索引及年龄段索引分别如表 12.8、表 12.9 和表 12.10 所示。

表 12.8 性别倒排索引

次关键字	物理地址
男	1,3,4,5,7
女	2,6

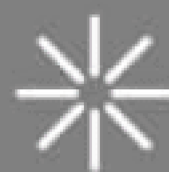
表 12.9 职称倒排索引

次关键字	物理地址
助教	5
讲师	3,7
副教授	1,2
教授	4,6

表 12.10 年龄倒排索引

次关键字	物理地址
20～29	3,5
30～39	1
40～49	2,7
50～59	4,6

12.3 补充练习题及参考答案



12.3.1 单项选择题

1. 影响文件检索效率的一个重要因素是________。

A. 逻辑记录的大小　　B. 物理记录的大小
C. 访问外存的次数　　D. 设备的读写速度

答：C。

2. 顺序文件适合于________。

A. 直接存取　　B. 成批处理
C. 按关键字存取　　D. 随机存取

答：顺序文件的优点是顺序存取速度较快,故顺序存取或成批处理较方便。本题的答案为 B。

3. 某索引文件的记录在逻辑上按关键字的顺序排列,但物理上不一定按关键字顺序存储,故需建立一个指示逻辑记录和物理记录之间一一对应关系的________。

A. 索引表　　B. 链接表　　C. 符号表　　D. 交叉访问表

答：A。

4. 在索引顺序文件中,________。

A. 主文件是无序的　　B. 主文件是有序的
C. 不适合随机查找　　D. 索引是稠密索引

答：B。

5. 索引非顺序文件是指________。

A. 主文件无序,索引表有序　　B. 主文件有序,索引表无序
C. 主文件有序,索引表有序　　D. 主文件无序,索引表无序

答：A。

6. 用 ISAM 和 VSAM 组织文件属于________。

A. 顺序文件　　B. 索引文件　　C. 哈希文件　　D. 多关键字文件

答：B。

7. 在删除 ISAM 文件记录时，一般________。

A. 只需做删除标志　　B. 需移动记录

C. 需改变指针　　D. 一旦删除就要做整理

答：ISAM 文件在删除记录时只需找到待删记录并置删除标志即可，并不需移动记录或改变指针。本题的答案为 A。

8. 直接存取方法是利用________进行组织的文件。

A. 哈希法　　B. 顺序索引法　　C. VSAM 法　　D. ISAM 法

答：直接存取文件也称哈希文件，是利用哈希方法进行组织的文件。本题的答案为 A。

9. 对于索引文件，稠密索引中的每个索引项对应被索引表中的________。

A. 所有记录　　B. n 条以下记录

C. 一条记录　　D. 多条记录

答：在索引文件中，若索引表中的每个索引项对应主文件中的一条记录，则称此索引表为稠密索引表。本题的答案为 C。

10. 对哈希文件进行直接存取的依据是________。

A. 按逻辑记录号去存取某个记录

B. 按逻辑记录的关键字去存取某个记录

C. 按逻辑记录的结构去存取某个记录

D. 按逻辑记录的具体内容去存取某个记录

答：B。

11. 直接存取文件的特点是________。

A. 记录按关键字排序　　B. 记录可以进行顺序存取

C. 存取速度快但占用较多的存储空间　　D. 记录不需要排序且存取效率高

答：直接存取文件即哈希文件，其特点是记录不需要排序且存取效率高。本题的答案为 D。

12. 对于哈希文件，以下说法错误的是________。

A. 哈希文件的插入、删除方便，不需要索引区且节省存储空间

B. 哈希文件只能按关键字随机存取且存取速度快

C. 经过多次插入、删除后可能出现溢出桶满而基桶内多数记录已被删除的情况

D. 哈希文件顺序存取方便

答：哈希文件的缺点是不能进行顺序存取，多次插入、删除后可能会造成文件结构不合理。本题的答案为 D。

13. 倒排文件的主要优点是________。

A. 便于进行插入和删除运算

B. 便于节省存储空间

C. 便于进行文件合并

D. 能大大提高基于非关键字的检索速度

答：D。

14. 倒排文件包含有若干个倒排表，倒排表的一个记录是________，倒排文件的检查速度快但修改、维护较难。

A. 一个主关键字值和该关键字的记录地址
B. 一个次关键字值和它对应的一个记录地址
C. 一个次关键字值和它对应的全部记录地址
D. 多个主关键字值和它们相对应的某个记录地址

答：C。

15. 倒排文件中的倒排表是指________。

A. 主关键字索引　　B. 次关键字索引
C. 物理顺序与逻辑顺序不一致　　D. 多关键字索引

答：B。

12.3.2 填空题

1. 文件中的记录有物理记录和________之分。

答：逻辑记录。

2. 索引顺序文件既可以顺序存取，也可以________存取。

答：直接。

3. 顺序文件是指记录按进入文件的先后顺序存放，其________相一致。

答：逻辑顺序和物理顺序。

4. 对文件的检索有___①___、___②___和___③___检索3种方式。

答：①顺序　②直接　③按关键字。

5. 索引文件由________和主文件两部分组成。

答：索引表。

6. 一个索引文件的索引表都是按________有序的。

答：关键字。

7. 索引文件的检索分成两步完成，第一步是___①___，第二步是___②___。

答：①将索引表读入内存查找到相应的物理地址　②根据索引表所指示的物理地址将记录所在的数据块读入内存进行查找。

8. 哈希文件是用________方法组织的。

答：哈希。

9. 在多重表文件中，每个索引表通常都是________。

答：稀疏索引。

12.3.3 判断题

判断以下叙述的正确性。

(1) 构成文件的基本单位是记录。
(2) 对顺序文件来说，两个相邻的逻辑记录一定是存放在两个相邻的物理记录中的。
(3) 索引非顺序文件的特点是索引表中的索引项不一定按关键字大小有序排列。
(4) 对索引文件来说，索引表是建立在内存的，数据区是建立在外存的。
(5) 非稠密索引只能用于索引顺序文件。
(6) 稠密索引的优点是节省存储空间。

(7) 哈希文件是一种直接存取文件。

(8) 在哈希文件上是不可以进行顺序存取的。

(9) 多重表文件中的次关键字索引是用链接方法组织起来的。

(10) 多重表文件一般含有多个索引表。

(11) 建立动态索引的目的是为了便于修改。

(12) B－树和 B＋树都是用来实现动态索引的。

答:(1) 正确。

(2) 正确。

(3) 错误。

(4) 错误。

(5) 正确。索引顺序文件通常采用稀疏索引。

(6) 错误。稠密索引通常占用较多的存储空间。

(7) 正确。

(8) 正确。

(9) 正确。

(10) 正确。

(11) 正确。

(12) 正确。

12.3.4 简答题

1. 常用的文件组织方式有哪几种,各有何特点?

答:常用的文件组织方式有下列 4 种。

(1) 顺序组织:记录的物理存放顺序与记录间的逻辑顺序完全一致的存储结构。按这种方式存储的文件就是"顺序文件",这种文件用于顺序存储和批处理。

(2) 索引组织:利用索引结构组织的文件有一个索引表,其中包括一组关键字和对应的记录地址。通常索引表是按关键字的升序排列的。一个关键字及其对应的记录地址称为"索引项"。如果文件中的每个记录对应一个索引项,这种索引称为"稠密索引"。如果文件的每个物理块对应一个索引项,则称这种索引为"稀疏索引"。通常,稀疏索引的索引项由物理块的起始地址和块中的最大关键字组成。当索引表很大时还需要对索引表再建索引,形成索引树,主要的索引树有 B－树和 B＋树。

(3) 哈希组织:即直接存取文件。选择一种函数,对记录的关键字进行转换,用所得的函数值作为存放该记录的地址。用这种方法组织的文件称为"哈希文件"。

(4) 链组织:链组织与内存中的链式存储方式相同。通常作为文件组织中的辅助存储方式,如哈希文件中的溢出处理、多重表文件中主记录的链接等都要用到链组织。

在文件组织中,存储结构并非是单一的一种形式,往往是多种方法的组合。

2. 简述多关键字文件的组成。

答:多关键字文件不仅对主关键字索引,还对其余的次关键字进行索引。多重表文件是对数据文件中的主关键字和次关键字分别建立索引,索引采用指针构成链表。

3. 假设某文件有 21 个记录,其记录关键字序列为(7,23,1,18,4,24,56,184,27,63,

35,109,15,26,83,215,19,8,16,33,75)。构造一个哈希文件,桶的大小 $m=3$,期望对文件进行一次查询时读取外存数的平均值不超过 1.5。试问该文件应有多大?用除余法作为哈希函数,请设计此函数并画出构造好的哈希文件。

答:已知记录个数 $n=21$,桶容量 $m=3$,存取桶数的期望值(即拉链法成功查找长度)ASL=1.5。

因为拉链法的 $\text{ASL}=1+\alpha/2=1.5$,求得 $\alpha=1$。又因为

$$\alpha=\frac{n}{b\times m}\quad (b\times m\text{ 为总长度})$$

所以桶数 $b=\frac{n}{\alpha\times m}=7$,可知本文件应有 7 个桶。

设计哈希函数为 $H(\text{key})=\text{key mod } 7$。

用此函数算出各个记录桶号后构成的哈希文件如图 12.2 所示。

桶编号	基桶				溢出桶			
0	7	56	63	→	35			∧
1	1	15	8	∧				
2	23	184	16	∧				
3	24			∧				
4	18	4	109	∧				
5	26	215	19	→	33	75		∧
6	27	83		∧				

图 12.2 哈希文件

4. 已知职工文件中包括职工号、职工姓名、职务和职称 4 个数据项,如表 12.11 所示。职务有"校长"、"系主任"、"室主任"和"教员";"校长"领导所有"系主任","系主任"领导他所在系的所有"室主任","室主任"领导他所在室的全部"教员";职称有教授、副教授和讲师 3 种。请在职工文件的数据结构中设置若干指针项和索引,以满足下列两种查找的需要:

(1)能够检索出全部职工之间各级领导与被领导的情况;

(2)能够分别检索出全部教授、全部副教授或者全部讲师。

表 12.11 职工文件

职工号	姓名	职务	职称
001	张军	教员	讲师
002	沈灵	系主任	教授
003	叶明	校长	教授
004	张莲	室主任	副教授
005	叶宏	系主任	教授
006	周芳	教员	教授
007	刘光	系主任	教授
008	黄兵	教员	讲师
009	李民	室主任	教授
010	赵松	教员	副教授

要求指针数量尽可能少，给出各指针项索引的名称及含义即可。

答：为了能够检索出全部职工之间各级领导与被领导的情况，在职务项中增加一个“领导”指针项，指向其领导者。例如，最高领导“校长”的该指针项为空（NULL），每个“系主任”的领导指针项指向“校长”，每个“室主任”的领导指针项指向相对应的“系主任”，每个“教员”的领导指针项指向相对应的“室主任”。这样可以方便查找诸如某个“教员”的“系主任”是谁。

另外，还需建立一个“被领导”指针项，指向同一被领导的下一个职工，并建立一个相对应的“被领导”索引表。例如，假设004号“室主任”领导两个教员为001和008，则004的“被领导”指针项指向001，001的“被领导”指针项指向008，008的“被领导”指针项为空。在“被领导”索引表中004对应的索引项的头指针为001。这样可以方便查找某领导所领导的所有下属职工。

为了能够分别检索出全部教授、全部副教授、全部讲师，在职称项中增加一个“职称”指针项和一个“职称”索引表。“职称”指针项指向同一职称的下一个职工，例如，对于“讲师”职称，001的“职称”指针项指向008，008的“职称”指针项为NULL。增加一个“职称”索引表如表12.12所示。

表12.12 “职称”索引表

关键字	头指针	长度
讲师	001	2
副教授	004	2
教授	002	6

附录 A　两份本科生期末考试试题

本科生期末考试试题 1

要求：所有题目的解答均写在答题纸上，需写清楚题目的序号。每张答题纸都要写上姓名和学号。

一、单项选择题（共 15 小题，每小题 2 分，共计 30 分）

1. 数据结构是指________。

A. 一种数据类型

B. 数据的存储结构

C. 一组性质相同的数据元素的集合

D. 相互之间存在一种或多种特定关系的数据元素的集合

2. 以下算法的时间复杂度为________。

```
void fun(int n)
{    int i = 1,s = 0;
     while (i <= n)
     {    s += i + 100; i++;   }
}
```

A. $O(n)$　　B. $O(\sqrt{n})$　　C. $O(n\log_2 n)$　　D. $O(\log_2 n)$

3. 在一个长度为 n 的有序顺序表中删除第一个元素值为 x 的元素时，在查找元素 x 时采用二分查找方法，此时删除算法的时间复杂度为________。

A. $O(n)$　　B. $O(n\log_2 n)$　　C. $O(n^2)$　　D. $O(\sqrt{n})$

4. 若一个栈采用数组 $s[0..n-1]$存放其元素，初始时栈顶指针为 n，则以下元素 x 进栈的操作正确的是________。

A. top++；s[top]=x；　　B. s[top]=x；top++；

C. top−−；s[top]=x；　　D. s[top]=x；top−−；

5. 设环形队列中数组的下标为 $0\sim N-1$，其队头、队尾指针分别为 front 和 rear(front 指向队列中队头元素的前一个位置，rear 指向队尾元素的位置)，则其元素个数为________。

A. rear−front　　B. rear−front−1

C. (rear−front)%N+1　　D. (rear−front+N)%N

6. 若用一个大小为 6 的数组来实现环形队列，队头指针 front 指向队列中队头元素的前一个位置，队尾指针 rear 指向队尾元素的位置。若当前 rear 和 front 的值分别为 0 和 3，当从队列中删除一个元素，再加入两个元素后，rear 和 front 的值分别为________。

A. 1 和 5　　B. 2 和 4　　C. 4 和 2　　D. 5 和 1

7. 一棵高度为 $h(h\geqslant 1)$ 的完全二叉树至少有________个结点。

A. 2^{h-1}　　B. 2^h　　C. 2^h+1　　D. $2^{h-1}+1$

8. 设一棵哈夫曼树中有 999 个结点，该哈夫曼树用于对________个字符进行编码。

A. 999　　B. 499　　C. 500　　D. 501

9. 一个含有 n 个顶点的无向连通图采用邻接矩阵存储，则该矩阵一定是________。

A. 对称矩阵　　B. 非对称矩阵　　C. 稀疏矩阵　　D. 稠密矩阵

10. 设无向连通图有 n 个顶点、e 条边，若满足________，则图中一定有回路。

A. $e\geqslant n$　　B. $e<n-1$　　C. $e=n-1$　　D. $2e\geqslant n$

11. 如果从无向图的任一顶点出发进行一次广度优先遍历即可访问所有顶点，则该图一定是________。

A. 完全图　　B. 连通图　　C. 有回路　　D. 一棵树

12. 设有 100 个元素的有序表，在用折半查找时，不成功查找时最大的比较次数是________。

A. 25　　B. 50　　C. 10　　D. 7

13. 从 100 个元素确定的顺序表中查找某个元素(关键字为正整数)，如果最多只进行 5 次元素之间的比较，则采用的查找方法只可能是________。

A. 折半查找　　B. 顺序查找

C. 哈希查找　　D. 二叉排序树查找

14. 有一个含有 $n(n>1000)$ 个元素的数据序列，某人采用了一种排序方法对其按关键字递增排序，该排序方法需要关键字比较，其平均时间复杂度接近最好的情况，空间复杂度为 $O(1)$，该排序方法可能是________。

A. 快速排序　　B. 堆排序

C. 二路归并排序　　D. 基数排序

15. 对一个线性序列进行排序，该序列采用单链表存储，最好采用________方法。

A. 直接插入排序　　B. 希尔排序

C. 快速排序　　D. 都不适合

二、问答题(共 3 小题，每小题 10 分，共计 30 分)

1. 如果对含有 $n(n>1)$ 个元素的线性表的运算只有 4 种：删除第一个元素；删除最后一个元素；在第一个元素前面插入新元素；在最后一个元素的后面插入新元素，则最好采用以下哪种存储结构，并简要说明理由。

(1) 只有尾结点指针没有头结点指针的循环单链表。

(2) 只有尾结点指针没有头结点指针的非循环双链表。

(3) 只有头结点指针没有尾结点指针的循环双链表。

(4) 既有头结点指针也有尾结点指针的循环单链表。

2. 对于一个带权连通无向图 G，可以采用 Prim 算法构造出从某个顶点 v 出发的最小生成树，问该最小生成树是否一定包含从顶点 v 到其他所有顶点的最短路径。如果回答是，请予以证明；如果回答不是，请给出反例。

3. 有一棵二叉排序树按先序遍历得到的序列为(12,5,2,8,6,10,16,15,18,20)。回答以下问题：

(1) 画出该二叉排序树。

(2) 给出该二叉排序树的中序遍历序列。

(3) 求在等概率下的查找成功和不成功情况下的平均查找长度。

三、算法设计题(共 3 小题,共计 40 分)

1. (15 分)假设二叉树 b 采用二叉链存储结构,设计一个算法 void findparent(BTNode $*b$,ElemType x,BTNode $*\&p$)求指定值为 x 的结点的双亲结点 p。提示,根结点的双亲为 NULL,若在二叉树 b 中未找到值为 x 的结点,p 也为 NULL。

2. (10 分)假设一个有向图 G 采用邻接表存储,设计一个算法判断顶点 i 和顶点 $j(i\neq j)$之间是否相互连通,假设这两个顶点均存在。

3. (15 分)有一个含有 n 个整数的无序数据序列,所有的数据元素均不相同,采用整数数组 $R[0..n-1]$存储,请完成以下任务:

(1) 设计一个尽可能高效的算法,输出该序列中第 $k(1\leqslant k\leqslant n)$小的元素,算法中给出适当的注释信息。提示,利用快速排序的思路。

(2) 分析你所设计的求解算法的平均时间复杂度,并给出求解过程。

本科生期末考试试题 1 参考答案

一、单项选择题(共 15 小题,每小题 2 分,共计 30 分)

1. D 2. B 3. A 4. C 5. D 6. B 7. A 8. C

9. A 10. A 11. B 12. D 13. C 14. B 15. A

二、问答题(共 3 小题,每小题 10 分,共计 30 分)

1. 采用存储结构(3),因为实现上述 4 种运算的时间复杂度均为 $O(1)$。

2. 不是。图 A.1 所示的图 G 从顶点 0 出发的最小生成树如图 A.2 所示,从顶点 0 到顶点 2 的最短路径为 0→2,而不是最小生成树中的 0→1→2。

3. (1) 先序遍历得到的序列为(12,5,2,8,6,10,16,15,18,20),中序序列是一个有序序列,所以为(2,5,6,8,10,12,15,16,18,20),由先序序列和中序序列可以构造出对应的二叉树,如图 A.3 所示。

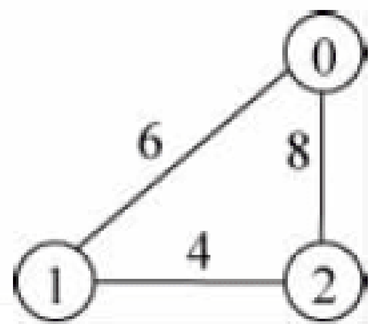


图 A.1 一个带权连通无向图 G

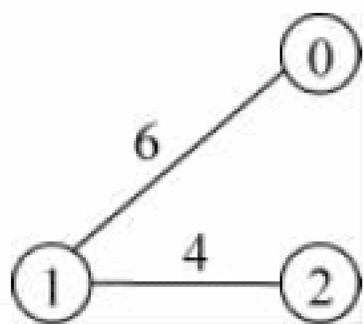


图 A.2 图 G 的一棵最小生成树

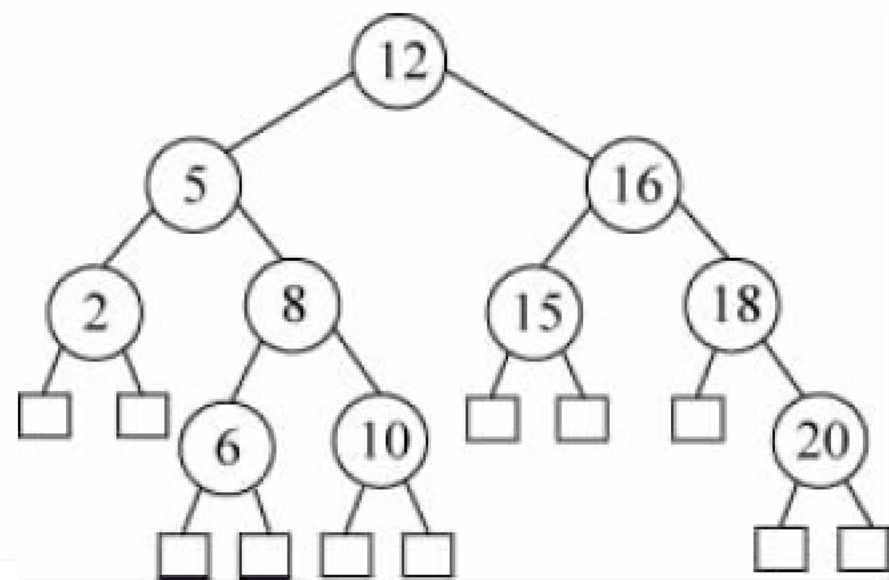


图 A.3 一棵二叉排序树

(2) 中序遍历序列为 2,5,6,8,10,12,15,16,18,20。

(3) $ASL_{成功}=(1\times1+2\times2+4\times3+3\times4)/10=29/10=2.9$。

$ASL_{不成功}=(5\times3+6\times4/11=39/11$。

三、算法设计题(共 3 小题,共计 40 分)

1. (15 分)参考算法如下:

```
void findparent(BTNode *b,ElemType x,BTNode *&p)    //求值为 x 的结点的双亲结点 p
{   if (b!= NULL)
    {   if (b->data==x) p=NULL;
        else if (b->lchild!= NULL && b->lchild->data==x)
            p=b;
        else if (b->rchild!= NULL && b->rchild->data==x)
            p=b;
        else
        {   findparent(b->lchild,x,p);
            if (p==NULL)
                findparent(b->rchild,x,p);
        }
    }
    else p=NULL;
}
```

2. (10 分)参考算法如下:

```
int visited[MAXV];                        //全局数组
void DFS(ALGraph *G,int v)                //深度优先遍历算法
{   ArcNode *p;
    visited[v]=1;                         //置已访问标记
    p=G->adjlist[v].firstarc;             //p指向顶点 v 的第一个邻接点
    while (p!= NULL)
    {   if (visited[p->adjvex]==0)        //若 p->adjvex 顶点未访问,递归访问它
            DFS(G,p->adjvex);
        p=p->nextarc;                     //p指向顶点 v 的下一个邻接点
    }
}
bool DFSTrave(ALGraph *G,int i,int j)    //判断顶点 i 和顶点 j(i≠j)之间是否相互连通
{   int k;
    bool flag1=false,flag2=false;
    for (k=0;k<G->n;k++)
        visited[k]=0;
    DFS(G,i);                             //从顶点 i 开始进行深度优先遍历
    if (visited[j]==1)
        flag1=true;
    for (k=0;k<G->n;k++)
        visited[k]=0;
    DFS(G,j);                             //从顶点 j 开始进行深度优先遍历
    if (visited[i]==1)
        flag2=true;
    if (flag1 && flage2)
        return true;
    else
        return false;
}
```

3. (15 分)(1) 采用快速排序的算法如下:

```
int QuickSelect(int R[],int s,int t,int k)      //在 R[s..t]序列中找第 k 小的元素
{   int i = s,j = t;
    int tmp;
    if (s < t)                                  //区间内至少存在两个元素的情况
    {   tmp = R[s];                             //用区间的第 1 个记录作为基准
        while (i!= j)                              //从区间两端交替向中间扫描,直到 i = j 为止
        {   while (j > i && R[j]> = tmp)
                j -- ;                          //从右向左扫描,找第 1 个小于 tmp 的 R[j]
            R[i] = R[j];                        //将 R[j]前移到 R[i]的位置
            while (i < j && R[i]< = tmp)
                i++;                            //从左向右扫描,找第 1 个大于 tmp 的 R[i]
            R[j] = R[i];                        //将 R[i]后移到 R[j]的位置
        }
        R[i] = tmp;
        if (k - 1 == i) return R[i];
        else if (k - 1 < i) return QuickSelect(R,s,i - 1,k);      //在左区间中递归查找
        else return QuickSelect(R,i + 1,t,k); //在右区间中递归查找
    }
    else if (s == t && s == k - 1)              //区间内只有一个元素且为 R[k - 1]
        return R[k - 1];
}
void Mink(int R[],int n,int k)                  //输出整数数组 R[0..n - 1]中第 k 小的元素
{   if (k >= 1 && k <= n)
        printf(" %d\n",QuickSelect(R,0,n - 1,k));
}
```

(2) 对于求 R 中第 k 小元素的算法 Mink(R,n,k),设算法平均执行时间为 $T(n)$,有以下递推式:

$$T(1)=1,\quad T(n)=T(n/2)+O(n)$$

则:

$$\begin{aligned}T(n)&=T(n/2)+O(n)=T(n/2^2)+O(n)+O(n/2)=\cdots\\&=T(n/2^m)+O(n)+O(n/2)+\cdots+O(n/2^m)\qquad //m=\log_2 n\\&=O(1)+O(n)+O(n)\\&=O(n)\end{aligned}$$

所以,该算法的平均时间复杂度为 $O(n)$。

本科生期末考试试题 2

要求:所有题目的解答均写在答题纸上,需写清楚题目的序号。每张答题纸都要写上姓名和学号。

一、单项选择题(每小题 1.5 分,20 小题,共计 30 分)

1. 以下数据结构中________属非线性结构。

A. 栈　　B. 串　　C. 队列　　D. 平衡二叉树

2. 以下算法的时间复杂度为________。

```
void func(int n)
{   int i = 0,s = 0;
    while (s <= n)
    {   i++;
        s = s + i;
    }
}
```

A. $O(n)$　　B. $O(\sqrt{n})$　　C. $O(n\log_2 n)$　　D. $O(\log_2 n)$

3. 在一个双链表中,删除 p 所指结点(非首、尾结点)的操作是________。

A. p->prior->next=p->next; p->next->prior=p->prior

B. p->prior=p->prior->prior; p->prior->prior=p

C. p->next->prior=p; p->next=p->next->next

D. p->next=p->prior->prior; p->prior=p->prior->prior

4. 设 n 个元素的进栈序列是 1、2、3、…、n,其输出序列是 p_1、p_2、…、p_n,若 $p_1=3$,则 p_2 的值为________。

A. 一定是 2　　B. 一定是 1　　C. 不可能是 1　　D. 以上都不对

5. 在数据处理过程中经常需要保存一些中间数据,如果要实现先保存的数据先处理,则应采用________来保存这些数据。

A. 线性表　　B. 栈　　C. 队列　　D. 单链表

6. 中缀表达式 $a*(b+c)-d$ 对应的后缀表达式是________。

A. $a\ b\ c\ d\ *\ +\ -$　　B. $a\ b\ c\ +\ *\ d\ -$

C. $a\ b\ c\ *\ +\ d\ -$　　D. $-\ +\ *\ a\ b\ c\ d$

7. 设栈 s 和队列 q 的初始状态都为空,元素 a、b、c、d、e 和 f 依次通过栈 s,一个元素出栈后即进入队列 q,若 6 个元素出队的序列是 b、d、c、f、e、a,则栈 s 的容量至少能存________个元素。

A. 2　　B. 3　　C. 4　　D. 5

8. 执行以下________操作时,需要使用队列作为辅助存储空间。

A. 图的深度优先遍历　　B. 二叉树的先序遍历

C. 平衡二叉树查找　　D. 图的广度优先遍历

9. 若将 n 阶上三角矩阵 **A** 按列优先顺序压缩存放在一维数组 $B[1..n(n+1)/2]$ 中,**A** 中第一个非零元素 $a_{1,1}$ 存于 B 数组的 b_1 中,则应存放到 b_k 中的元素 $a_{i,j}(1\leqslant i\leqslant j)$ 的下标 i、j 与 k 的对应关系是________。

A. $i(i+1)/2+j$　　B. $i(i-1)/2+j$

C. $j(j+1)/2+i$　　D. $j(j-1)/2+i$

10. 一棵结点个数为 n、高度为 h 的 $m(m\geqslant 3)$ 次树中,其总分支数是________。

A. nh　　B. n+m　　C. $n-1$　　D. $h-1$

11. 设森林 F 对应的二叉树为 B,B 中有 m 个结点,其根结点的右子树的结点个数为 n,森林 F 中第一棵树的结点个数是________。

A. $m-n$　　B. $m-n-1$

C. $n+1$　　D. 条件不足，无法确定

12. 一棵二叉树的先序遍历序列为 ABCDEF、中序遍历序列为 CBAEDF，则后序遍历序列为________。

A. CBEFDA　　B. FEDCBA　　C. CBEDFA　　D. 不确定

13. 在一个具有 n 个顶点的有向图中，构成强连通图时至少有________条边。

A. n　　B. $n+1$　　C. $n-1$　　D. $n/2$

14. 对于有 n 个顶点的带权连通图，它的最小生成树是指图中任意一个________。

A. 由 $n-1$ 条权值最小的边构成的子图

B. 由 $n-1$ 条权值之和最小的边构成的子图

C. 由 $n-1$ 条权值之和最小的边构成的连通子图

D. 由 n 个顶点构成的极小连通子图，且边的权值之和最小

15. 对于有 n 个顶点、e 条边的有向图，采用邻接矩阵表示，求单源最短路径的 Dijkstra 算法的时间复杂度为________。

A. $O(n)$　　B. $O(n+e)$　　C. $O(n^2)$　　D. $O(ne)$

16. 一棵高度为 h 的平衡二叉树，其中每个非叶子结点的平衡因子均为 0，则该树的结点个数是________。

A. $2^{h-1}-1$　　B. 2^{h-1}　　C. $2^{h-1}+1$　　D. 2^h-1

17. 在对线性表进行折半查找时，要求线性表必须________。

A. 以顺序方式存储

B. 以链接方式存储

C. 以顺序方式存储，且结点按关键字有序排序

D. 以链表方式存储，且结点按关键字有序排序

18. 假设有 k 个关键字互为同义词，若用线性探测法把这 k 个关键字存入哈希表中，至少要进行________次探测。

A. $k-1$　　B. k　　C. $k+1$　　D. $k(k+1)/2$

19. 在以下排序算法中，某一趟排序结束后未必能选出一个元素放在其最终位置上的是________。

A. 堆排序　　B. 冒泡排序

C. 直接插入排序　　D. 快速排序

20. 在以下排序方法中，________不需要进行关键字的比较。

A. 快速排序　　B. 归并排序　　C. 基数排序　　D. 堆排序

二、问答题（共 4 小题，每小题 10 分，共计 40 分）

1. 已知一棵度为 m 的树中有 n_1 个度为 1 的结点、n_2 个度为 2 的结点、…、n_m 个度为 m 的结点，问该树中有多少个叶子结点？（需要给出推导过程）

2. 设关键字序列 $D=(1,12,5,8,3,10,7,13,9)$，试完成下列各题：

(1) 依次取 D 中的各关键字，构造一棵二叉排序树 bt。

(2) 如何依据此二叉树 bt 得到 D 的一个关键字递增序列。

(3) 画出在二叉树 bt 中删除 12 后的树结构。

3. 一个有 $n(n>10)$ 个整数的数组 $R[1..n]$，其中所有元素是有序的，将其看成是一棵完全二叉树，该树构成了一个堆吗？若不是，请给一个反例；若是，请简要说明理由。

4. 若要在 n 个海量数据（超过十亿，不能一次全部放入内存）中找出最大的 k 个数（内存可以容纳 k 个数），最好采用什么数据结构和策略？请详细说明你采用的数据结构和策略，并用时间复杂度和空间复杂度来说明理由。

三、算法设计题（共计 30 分）

1. 设 $A=(a_1,a_2,\cdots,a_n)$，$B=(b_1,b_2,\cdots,b_m)$ 是两个递增有序的线性表（其中 n、m 均大于 1），且所有数据元素均不相同。假设 A、B 均采用带头结点的单链表存放，设计一个尽可能高效的算法判断 B 是否为 A 的一个连续子序列，并分析你设计的算法的时间复杂度和空间复杂度。（15 分）

2. 假设二叉树 b 采用二叉链存储结构存储，试设计一个算法，求该二叉树中从根结点出发的一条最长的路径长度，并输出此路径上各结点的值。（15 分）

本科生期末考试试题 2 参考答案

一、单项选择题（共 20 小题，每小题 1.5 分，共计 30 分）

1. D　2. B　3. A　4. C　5. C　6. B　7. B　8. D
9. D　10. C　11. A　12. A　13. A　14. D　15. C　16. D
17. C　18. D　19. C　20. C

二、问答题（共 4 小题，每小题 10 分，共计 40 分）

1. 依题意，设 n 为总的结点个数，n_0 为叶子结点（即度为 0 的结点）的个数，则有 $n=n_0+n_1+n_2+\cdots+n_m$。

又有 $n-1=$度的总数，即 $n-1=n_1\times1+n_2\times2+\cdots+n_m\times m$。

两式相减得 $1=n_0-n_2-2n_3-\cdots-(m-1)n_m$。

则有 $n_0=1+n_2+2n_3+\cdots+(m-1)n_m=1+\sum_{i=2}^{m}(i-1)n_i$。

2. (1) 本题构造的二叉排序树如图 A.4 所示。

(2) D 的有序序列为 bt 的中序遍历次序，即 1、3、5、7、8、9、10、12、13。

(3) 为了删除结点 12，找到其左子树中的最大结点 10（其双亲结点为 8），将该结点删除并用 10 代替 12，删除后的树结构如图 A.5 所示。

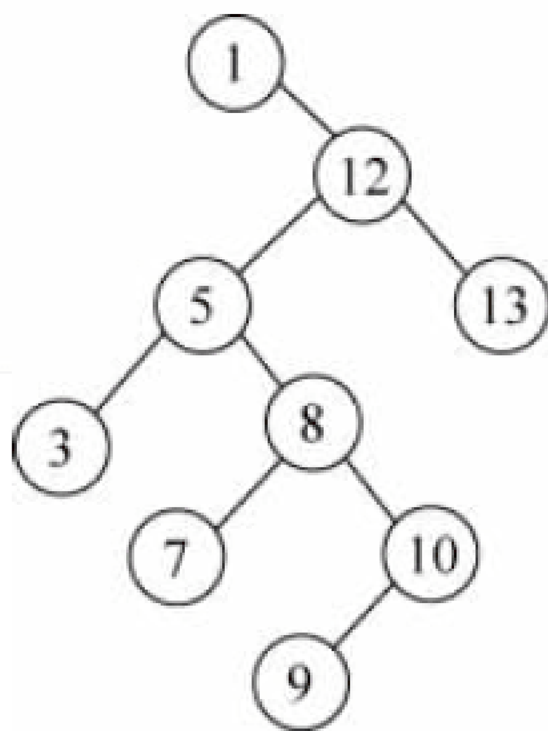


图 A.4　一棵二叉排序树

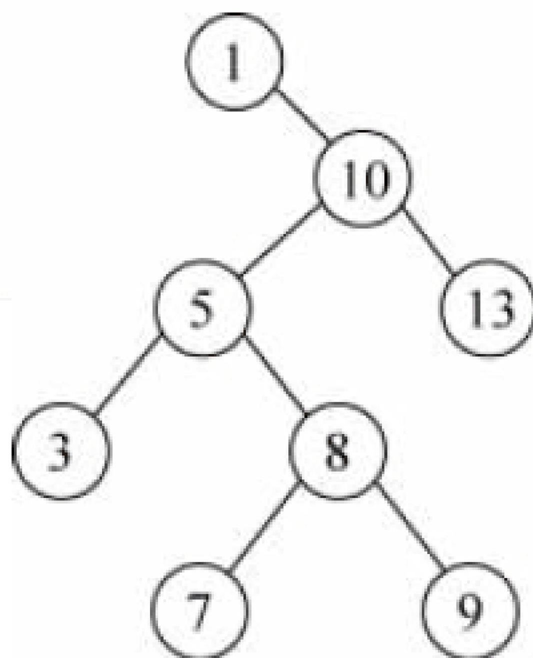


图 A.5　删除 12 后的二叉排序树

3. 该数组一定构成一个堆，递增有序数组构成一个小根堆，递减有序数组构成一个大根堆。

以递增有序数组为例，假设数组元素为 k_1、k_2、…、k_n 是递增有序的，从中看出下标越大的元素值也越大，对于任一元素 k_i 有 $k_i<k_{2i}$，$k_i<k_{2i+1}$（$i<n/2$），这正好满足小根堆的特性，所以构成一个小根堆。

4. 首先读入 k 个数，假设第一次读取的 k 个数就是前 k 个最大的数，把 k 个数建成小顶堆。然后从第 $k+1$ 个数开始，每个数都与堆顶的数值进行比较，如果数字 d 大于堆顶元素，则把堆顶的元素替换成 d，再将其调整成为小顶堆。当所有数据都读入并比较完之后，这个小顶堆里面的所有元素就是最大的 k 个数。其时间复杂度为 $O(n\log_2 k)$、空间复杂度为 $O(k)$。

三、算法设计题（共计 30 分）

1. （15 分）采用二路归并思路，用 p、q 分别扫描有序单链表 A、B，先找到第一个两者值相等的结点，然后在两者值相等时同步后移，如果 B 扫描完毕返回 true，否则返回 false。对应的算法如下：

```
bool SubSeq(LinkList *A,LinkList *B)
{   LinkList *p = A->next, *q = B->next;
    while (p!= NULL && q!= NULL)          //找两个单链表中第一个值相同的结点
    {   if (p->data<q->data)
            p = p->next;
        else if (p->data>q->data)
            q = q->next;
        else
            break;
    }
    while (p!= NULL && q!= NULL && p->data == q->data)
    {   //当两者值相等时同步后移
        p = p->next;
        q = q->next;
    }
    if (q == NULL)                        //当 B 中结点比较完毕后返回 true
        return true;
    else                                  //否则返回 false
        return false;
}
```

本算法的时间复杂度为 $O(m+n)$、空间复杂度为 $O(1)$，其中 m、n 分别为 A、B 单链表的长度。

2. （15 分）有多种解法，参考答案采用层次遍历（采用非环形队列）：

```
void MaxPath(BTNode *b,ElemType maxpath[],int &maxpathlen)
//求出的最长路径是 maxpath[maxpathlen-1]
{   //maxpathlen 的初值为 0
    struct snode
    {   BTNode *node;                      //存放当前结点指针
```

```
        int parent;                          //存放双亲结点在队列中的位置
    } Qu[MaxSize];                           //定义非环形队列
    ElemType path[MaxSize];                  //存放一条路径
    int pathlen;                             //存放一条路径的长度
    int front,rear,p,i;                      //定义队头和队尾指针
    front = rear = -1;                       //置队列为空队列
    rear++;
    Qu[rear].node = b;                       //根结点指针进队
    Qu[rear].parent = -1;                    //根结点没有双亲结点
    while (front < rear)                     //队列不为空
    {   front++; b = Qu[front].node;         //队头出队列
        if (b->lchild == NULL && b->rchild == NULL)    //b 为叶子结点
        {   p = front;pathlen = 0;
            while (Qu[p].parent!= -1)
            {   path[pathlen] = Qu[p].node->data;
                pathlen++;
                p = Qu[p].parent;
            }
            path[pathlen] = Qu[p].node->data;
            pathlen++;
            if (pathlen > maxpathlen)        //通过比较求最长路径
            {   for (i = 0;i < pathlen;i++)
                    maxpath[i] = path[i];
                maxpathlen = pathlen;
            }
        }
        if (b->lchild!= NULL)                //左孩子结点进队
        {   rear++;
            Qu[rear].node = b->lchild;
            Qu[rear].parent = front;
        }
        if (b->rchild!= NULL)                //右孩子结点进队
        {   rear++;
            Qu[rear].node = b->rchild;
            Qu[rear].parent = front;
        }
    }
}
```

本算法的时间复杂度为 $O(n)$、空间复杂度为 $O(n)$。

附录 B　两份研究生入学考试(单考)数据结构部分试题

研究生入学考试(单考)数据结构部分试题 1

一、单项选择题(共 11 小题,每小题 2 分,共 22 分)

1. 以下叙述中正确的是________。

Ⅰ. 对于同一种逻辑结构,可以有多种逻辑结构表示方法

Ⅱ. 对于同一种逻辑结构,同一个运算在不同的存储方式下实现,其运算效率可能不同

Ⅲ. 在设计某种逻辑结构的存储结构时主要考虑的是存储数据元素

Ⅳ. 对于一种逻辑结构,可以采用多种存储结构进行存储

A. 仅Ⅰ、Ⅱ、Ⅲ　　B. 仅Ⅱ、Ⅲ、Ⅳ

C. 仅Ⅰ、Ⅱ、Ⅳ　　D. Ⅰ、Ⅱ、Ⅲ、Ⅳ

2. 顺序表具有随机存取特性指的是________。

A. 查找值为 x 的元素与顺序表中元素的个数 n 无关

B. 查找值为 x 的元素与顺序表中元素的个数 n 有关

C. 查找序号为 i 的元素与顺序表中元素的个数 n 无关

D. 查找序号为 i 的元素与顺序表中元素的个数 n 有关

3. 在一个算法中需要建立 $n(n\geqslant 3)$个栈时可以选择下列 3 种方案之一,对以下各种解决方案进行比较,其中错误的是________。

Ⅰ. 分别用多个顺序存储空间建立多个独立的栈

Ⅱ. 多个栈共享一个顺序存储空间

Ⅲ. 分别建立多个独立的链栈

A. Ⅰ方案的优点是操作简便　　B. Ⅱ方案的缺点是时间效率低

C. Ⅲ方案的优点是存储效率高　　D. 以上都不对

4. 某环形队列的元素类型为 char,队头指针 front 指向队头元素的前一个位置,队尾指针 rear 指向队尾元素,如图 B.1 所示,则队中元素为________。

A. abcd123456　　B. abcd123456c　　C. dfgbca　　D. cdfgbca

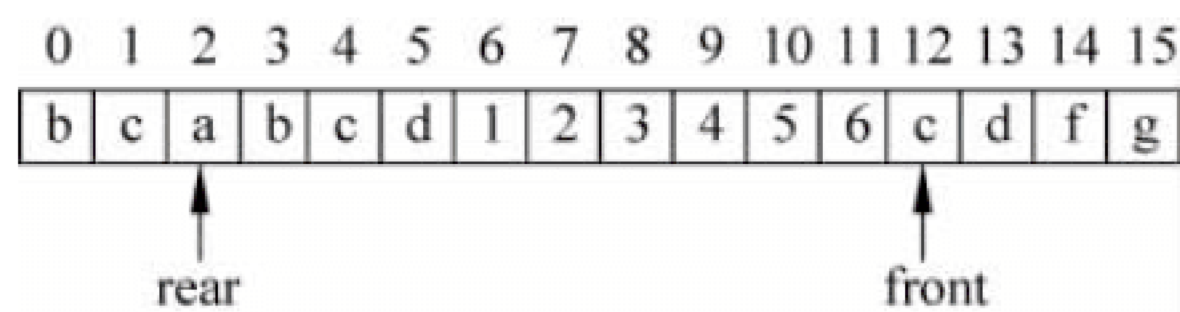


图 B.1　一个环形队列

5. 一个对称矩阵 $\mathbf{A}[1..10,1..10]$采用压缩存储方式,将其下三角部分按行优先存储到一维数组 $\boldsymbol{B}[1..m]$中,则 $\mathbf{A}[8][5]$元素在 B 中的位置 k 是________。

A. 33　　B. 37　　C. 45　　D. 60

6. 在高度为 $h(h\geqslant1)$ 的哈夫曼树中，最少有__①__个结点，最多有__②__个结点。

A. 2^h-1　　B. 2^{h-1}　　C. $2h$　　D. $2h-1$

7. 用 Dijkstra 算法求一个带权有向图 G 中从顶点 0 出发的最短路径，在算法执行的某时刻，$S=\{0,2,3,4\}$，选取的目标顶点是顶点 1，则可能修改的最短路径是________。

A. 从顶点 0 到顶点 2 的最短路径

B. 从顶点 2 到顶点 4 的最短路径

C. 从顶点 0 到顶点 1 的最短路径

D. 从顶点 0 到顶点 3 的最短路径

8. 在有向图 G 的拓扑序列中，若顶点 i 在顶点 j 之前，则以下情况不可能出现的是________。

A. G 中有边 $<i,j>$　　B. G 中有一条从顶点 i 到顶点 j 的路径

C. G 中没有边 $<i,j>$　　D. G 中有一条从顶点 j 到顶点 i 的路径

9. 在二叉排序树中，最小关键字结点的________。

A. 左指针一定为空　　B. 右指针一定为空

C. 左、右指针均为空　　D. 左、右指针均不空

10. 数据序列(5,4,15,10,3,2,9,6,11)是某排序方法第一趟后的结果，该排序算法可能是________。

A. 冒泡排序　　B. 二路归并排序

C. 堆排序　　D. 简单选择排序

11. 设有 1000 个无序的元素，希望用最快的速度挑选出其中前 10 个最大的元素，最好选用________法。

A. 冒泡排序　　B. 快速排序　　C. 堆排序　　D. 基数排序

二、综合应用题(共两小题，共 23 分)

1. (13 分)一棵二叉树采用二叉链存储结构存放，结点类型如下：

```
typedef struct node
{   ElemType data;
    struct node * lchild, * rchild;
} BTNode;
```

假设所有结点 data 域均不相同，设计一个算法删除并释放其中 data 域为 x 的结点及其子孙结点，算法中给出必要的注释。

2. (10 分)有一个含有 n 个元素的学生成绩线性表，每个元素含有姓名和分数(百分制)，成绩等级划分的标准是分数大于等于 90 为 A 类，不及格为 C 类，其余为 B 类。学生成绩线性表采用带头结点的单链表存储，结点类型如下：

```
typedef struct node
{   char name[10];                  //姓名
    int score;                      //分数
```

```
    struct node * next;
} LinkNode;
```

设计一个在时间和空间两方面尽可能高效的算法，使该学生成绩单链表按成绩等级 A、B、C 的次序排列，并给出你设计的算法的时间和空间复杂度，算法中给出必要的注释。

研究生入学考试(单考)数据结构部分试题 1 参考答案

一、单项选择题（共 11 小题，每小题 2 分，共 22 分）

1. C　2. C　3. C　4. C　5. A　6. ①D　②A
7. C　8. D　9. A　10. B　11. C

二、综合应用题（共两小题，共 23 分）

1. (13 分)参考算法如下：

```
BTNode * Find(BTNode *&b,char x)
//查找值为 x 的结点并返回,若不为根结点,将其双亲的相应指针域置为 NULL
{   BTNode * p;
    if (b== NULL)
        return NULL;
    else if (b->data== x)
        return b;
    else if (b->lchild!= NULL && b->lchild->data== x)
    {   p= b->lchild;
        b->lchild= NULL;
        return p;
    }
    else if (b->rchild!= NULL && b->rchild->data== x)
    {   p= b->rchild;
        b->rchild= NULL;
        return p;
    }
    else
    {   p= Find(b->lchild,x);
        if (p!= NULL)
            return p;
        else
            return Find(b->rchild,x);
    }
}
void Release(BTNode *&b)          //释放以 b 为根结点的子树
{   if (b!= NULL)
    {   Release(b->lchild);
        Release(b->rchild);
        free(b);
    }
}
void Delete(BTNode *&b,char x)   //删除并释放值 x 的结点的子树
```

```
{   BTNode *p;
    p=Find(b,x);
    if (p!=NULL)
        Release(p);
}
```

评分说明：Find 算法占 6 分(如果仅有查找没有将 x 结点的双亲相应指针置为 NULL，扣 1 分)，释放子树的 Release 算法占 4 分，Delete 算法占 3 分。

2. (10 分)参考算法如下：

```
void Rearrange(LinkNode *&L)
{   LinkNode *p, *La, *Lb, *Lc, *ra, *rb, *rc;
    p=L->next;
    La=Lb=Lc=NULL;                  //3 个不带头结点的单链表
    while (p!=NULL)
    {   if (p->score>=90)           //将 A 等成绩的结点链到 La 中
        {   if (La==NULL)
                La=p;
            else
                ra->next=p;
            ra=p; p=p->next;
        }
        else if (p->score<60)       //将 C 等成绩的结点链到 Lc 中
        {   if (Lc==NULL)
                Lc=p;
            else
                rc->next=p;
            rc=p; p=p->next;
        }
        else                        //将 B 等成绩的结点链到 Lb 中
        {   if (Lb==NULL)
                Lb=p;
            else
                rb->next=p;
            rb=p; p=p->next;
        }
    }
    L->next=La;                     //将 3 个单链表按等级 A、B、C 链起来
    ra->next=Lb;
    rb->next=Lc;
    rc->next=NULL;
}
```

算法的时间复杂度为 $O(n)$、空间复杂度为 $O(1)$。

评分说明：算法占 8 分，算法的时间和空间复杂度各占 1 分。如果设计的算法的时间、空间复杂度均为 $O(n)$，扣 3 分。如果算法不是最优，可适当给分，只要算法的时空分析正确，无论算法是否正确，时空分析部分仍给 2 分。

研究生入学考试(单考)数据结构部分试题 2

一、单项选择题(共 11 小题,每小题 2 分,共 22 分)

1. 设 n 是描述问题规模的非负整数,以下算法的时间和空间复杂度分别为________。

```
int fun(int n)
{   int i = 1,s = 1;
    while (i <= n)
    {   s++;
        i = 3 * i;
    }
    return s;
}
```

A. $O(\log_3 n)$、$O(n)$　　B. $O(\log_2 n)$、$O(1)$

C. $O(\log_2 n)$、$O(n)$　　D. $O(n\log_3 n)$、$O(1)$

2. 设有 5 个元素的进栈序列是 a、b、c、d、e,其出栈序列是 c、e、d、b、a,则该栈的容量至少是________。

A. 2　　B. 3　　C. 4　　D. 5

3. 以下各链表均不带有头结点,其中最不适合用作链栈的链表是________。

A. 只有表头指针没有表尾指针的循环双链表

B. 只有表尾指针没有表头指针的循环双链表

C. 只有表尾指针没有表头指针的循环单链表

D. 只有表头指针没有表尾指针的循环单链表

4. 设环形队列中数组的下标是 $0\sim N-1$,其队头、队尾指针分别为 f 和 r(f 指向队首元素的前一位置,r 指向队尾元素),已知 f 和队列中的元素个数 c,则 r 为________。

A. $(f+c)\%N$　　B. $(f-c)\%N$

C. $(f-c+N)\%N$　　D. $f+c$

5. 一个 10 阶对称矩阵 $\mathbf{A}[1..10,1..10]$采用压缩存储方式,将其上三角和主对角部分按行优先存储到一维数组 $B[1..m]$中,则 $\mathbf{A}[8][5]$元素值在 B 中的存储位置 k 是________。

A. 10　　B. 37　　C. 45　　D. 60

6. 一棵度为 5、结点个数为 20 的树,其高度的范围是________。

A. 3～20　　B. 5～18　　C. 3～16　　D. 3～4

7. 如果从某个非空无向图的任一顶点出发进行一次深度优先遍历即可访问所有顶点,则该图一定是________。

A. 完全图　　B. 连通图　　C. 有回路　　D. 一棵树

8. Dijkstra 算法是________方法求出图中从初始点到其余顶点的最短路径的。

A. 按长度递减的顺序求出图的初始点到其余顶点的最短路径

B. 按长度递增的顺序求出图的初始点到其余顶点的最短路径

C. 通过深度优先遍历求出图中初始点到其余顶点的最短路径

D. 通过广度优先遍历求出图中初始点到其余顶点的最短路径

9. 按关键字 13、24、37、90、53 的次序构造一棵平衡二叉树，该平衡二叉树的高度是________。

A. 3　　B. 4　　C. 5　　D. 不确定

10. 当一组待排序的数据已基本有序时，采用快速排序时的时间性能与________方法接近。

A. 希尔排序　　B. 堆排序

C. 二路归并排序　　D. 简单选择排序

11. 有 $n(n>100)$ 个十进制正整数进行基数排序，其中最大的整数为 5 位，则基数排序过程中临时建立的队数个数是________。

A. 10　　B. n　　C. 5　　D. 以上都不对

二、综合应用题（共两小题，共 23 分）

1. （12 分）假设一个学生年级有若干个班，每个班有唯一的班号（如 1、2 等），一个班有若干个学生，每个学生信息包括学号和姓名（同一个班的学号唯一，不同班的学号可能重复），学生记录按时间先后顺序插入。其中最频繁的操作如下：

① 删除某班某学号的学生记录。

② 在某班中插入一个某学号的学生记录。

③ 查找某班某学号的学生记录。

回答以下问题：

（1）设计一个你认为最合适的存储结构用于存储该年级的所有学生信息，并画出相应的示意图。

（2）给出在你设计的存储结构下实现上述操作②的过程，并说明其时间复杂度（用文字叙述即可，不必考虑插入学号与该班中其他记录的学号重复的情况）。

2. （11 分）一个含有 $n(n>10)$ 个整数的序列可以看成是一棵完全二叉树，该树采用二叉链存储结构存储，每个结点存放一个整数，根结点指针为 b，结点类型定义如下：

```
typedef struct node
{     int data;
      struct node * lchild, * rchild;
} BTNode;
```

设计一个算法判断该序列是否为一个大根堆，如果是一个大根堆，返回 true，否则返回 false，算法中给出适当的注释。

研究生入学考试(单考)数据结构部分试题 2 参考答案

一、单项选择题（共 11 小题，每小题 2 分，共 22 分）

1. B　2. C　3. D　4. A　5. B　6. C

7. B　8. B　9. A　10. D　11. A

二、综合应用题（共两小题，共 23 分）

1. （12 分）（1）每个学生信息用一个结点存储，一个班的学生信息构成一个带头结点的单链表，头结点包含班号，所有班的头结点构成一个单链表，用其头指针标识整个存储结构。

其示意图如图 B.2 所示。

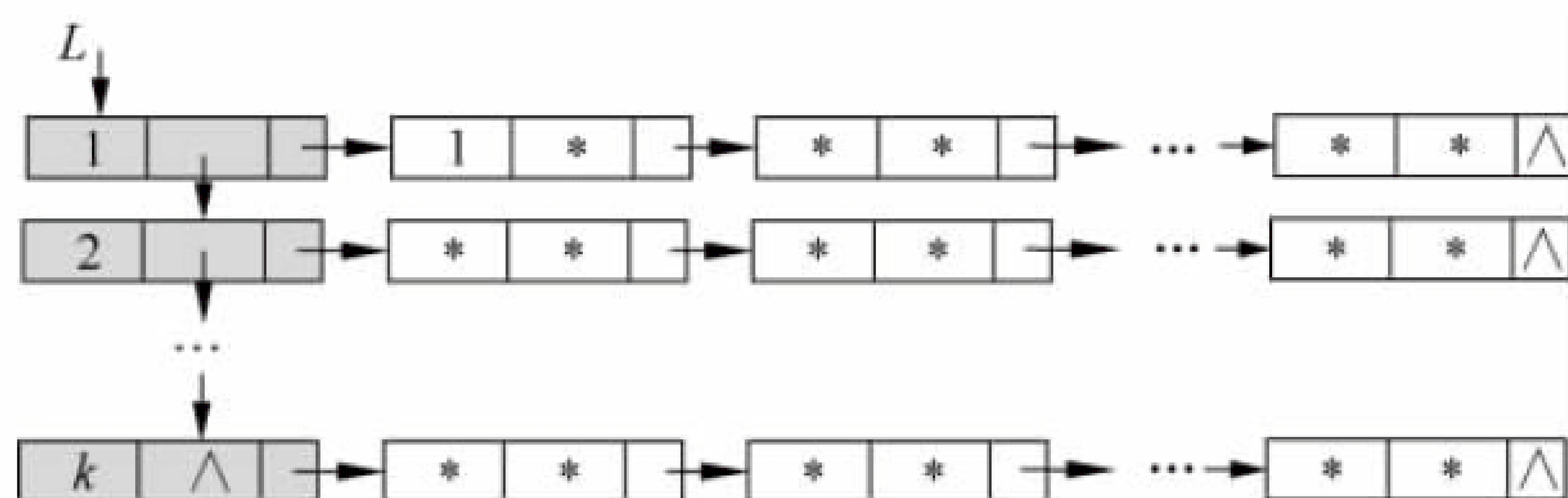


图 B.2　学生信息的存储结构

(2) 创建一个插入学生信息的结点,通过 L 找到指定班的头结点,在其后插入该结点。时间复杂度为 $O(1)$。

【评分说明】(1) 满分为 6 分,如果采用一个单链表存储所有学生信息,最多 4 分;如果采用数组存储所有学生信息,最多 2 分。

(2) 满分为 6 分,根据学生自己设计的存储结构来判分,时间复杂度占 3 分。

2. (11 分)参考算法如下:

```
bool isheap(BTNode * b)                          //判断一棵完全二叉树是否为大根堆
{   if (b!= NULL)
    {   if (b->lchild == NULL && b->rchild == NULL)//一个结点是一个堆
            return true;
        else
        {   if (b->rchild!= NULL)                //如果左、右子树不空
            {   if (b->data > b->lchild->data && b->data > b->rchild->data)
                                                 //根结点满足堆定义
                {   if (isheap(b->lchild) && isheap(b->rchild))
                                                 //如果左、右子树满足堆定义,返回 true
                        return true;
                    else return false;           //否则返回 false
                }
                else return false;               //根结点不满足堆定义返回 false
            }
            else                                 //如果左子树不空
            {   if (b->data > b->lchild->data)   //根结点满足堆定义
                {   if (isheap(b->lchild))       //如果左子树满足堆定义,返回 true
                        return true;
                    else return false;           //否则返回 false
                }
                else return false;               //根结点不满足堆定义返回 false
            }
        }
    }
    else return true;
}
```

【评分说明】　算法满分为 11 分,如果转换为数组再判断,最多 8 分。可以使用层次遍历来实现。

附录 C　两份全国计算机学科专业考研题数据结构部分试题

2014 年试题

一、单项选择题（每小题 2 分，只有一个选项是符合题目要求的）

1. 下列程序段的时间复杂度是________。

```
count = 0;
for(k = 1;k <= n;k * = 2)
    for(j = 1;j <= n;j++)
        count++;
```

A. $O(\log_2 n)$　　B. $O(n)$　　C. $O(n\log_2 n)$　　D. $O(n^2)$

2. 假设栈初始为空，在将中缀表达式 a/b+(c*d−e*f)/g 转换为等价的后缀表达式的过程中，当扫描到 f 时，栈中的元素依次是________。

A. +(*−　　B. +(−*　　C. /+(*−*　　D. /+−*

3. 循环两列放在一维数组 $A[0\cdots M-1]$中，end_1 指向队头元素，end_2 指向队尾元素的后一个位置。假设队列两端均可进行入队和出队操作，队列中最多能容纳 $M-1$ 个元素。初始时为空，下列判断队空和队满的条件中正确的是________。

A. 队空：$end_1 == end_2$；队满：$end_1 == (end_2+1) \bmod M$

B. 队空：$end_1 == end_2$；队满：$end2 == (end_1+1) \bmod (M-1)$

C. 队空：$end_2 == (end_1+1) \bmod M$；队满：$end_1 == (end_2+1) \bmod M$

D. 队空：$end_1 == (end_2+1) \bmod M$；队满：$end_2 == (end_1+1) \bmod (M-1)$

4. 若对如图 C.1 所示的二叉树进行中序线索化，则结点 x 的左、右线索指向的结点分别是________。

A. e,c　　B. e,a

C. d,c　　D. b,a

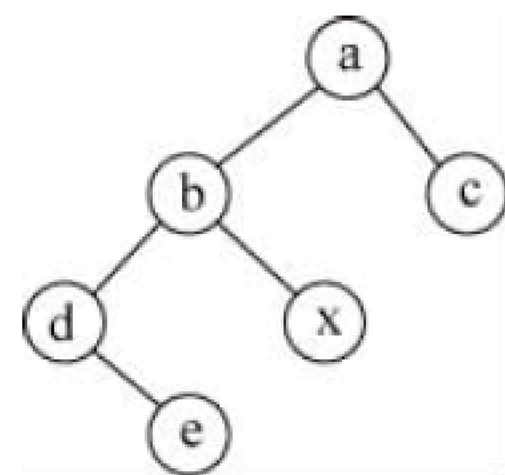


图 C.1　一棵二叉树

5. 将森林 F 转换为对应的二叉树 T，F 中叶子结点的个数等于________。

A. T 中叶子结点的个数

B. T 中度为 1 的结点个数

C. T 中左孩子指针为空的结点个数

D. T 中右孩子指针为空的结点个数

6. 5 个字符有以下 4 种编码方案，不是前缀编码的是________。

A. 01,0000,0001,001,1　　B. 011,000,001,010,1

C. 000,001,010,011,100　　D. 0,100,110,1110,1100

7. 对如图 C.2 所示的有向图进行拓扑排序，得到的拓扑序列可能是________。

A. 3,1,2,4,5,6　　B. 3,1,2,4,6,5

C. 3,1,4,2,5,6　　D. 3,1,4,2,6,5

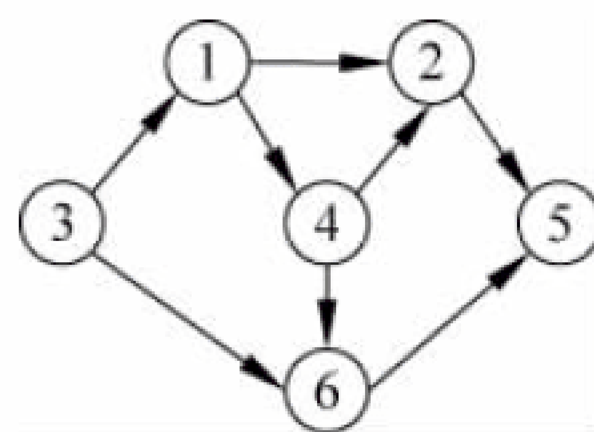


图 C.2　一个有向图

8. 用哈希(散列)方法处理冲突(碰撞)时可能出现堆积(聚集)现象，下列选项中会受堆积现象直接影响的是________。

A. 存储效率　　B. 散列函数

C. 装填(装载)因子　　D. 平均查找长度

9. 在一棵具有 15 个关键字的 4 阶 B－树中，含关键字的结点数最多是________。

A. 5　　B. 6　　C. 10　　D. 15

10. 在用希尔排序方法对一个数据序列进行排序时，若第 1 趟排序结果为(9,1,4,13,7,8,20,23,15)，则该趟排序采用的增量(间隔)可能是________。

A. 2　　B. 3　　C. 4　　D. 5

11. 在下列选项中，不可能是快速排序第 2 趟排序结果的是________。

A. 2,3,5,4,6,7,9　　B. 2,7,5,6,4,3,9

C. 3,2,5,4,7,6,9　　D. 4,2,3,5,7,6,9

二、综合应用题(共两小题，共 23 分)

1. (13 分)二叉树的带权路径长度(WPL)是二叉树中所有叶子结点的带权路径长度之和，给定一棵二叉树 T，采用二叉链表存储，结点结构为(left，weight，right)，其中叶子结点的 weight 域保存该结点的非负权值。设 root 为指向 T 的根结点的指针，设计求 T 的 WPL 的算法。要求：

(1) 给出算法的基本设计思想；

(2) 使用 C 或 C++ 语言，给出二叉树结点的数据类型定义；

(3) 根据设计思想，采用 C 或 C++ 语言描述算法，关键之处给出注释。

2. (10 分)某网络中的路由器运行 OSPF 路由协议，表 C.1 是路由器 R_1 维护的主要链路状态信息(LSI)，图 C.3 是根据表 C.1 及 R_1 的接口名构造出来的网络拓扑。

表 C.1　*R1* 所维护的 LSI

		R_1 的 LSI	R_2 的 LSI	R_3 的 LSI	R_4 的 LSI	备　注
Router ID		10.1.1.1	10.1.1.2	10.1.1.5	10.1.1.6	标识路由器的 IP 地址
Link1	ID	10.1.1.2	10.1.1.1	10.1.1.6	10.1.1.5	所连路由器的 RounterID
	IP	10.1.1.1	10.1.1.2	10.1.1.5	10.1.1.6	Link1 的基本 IP 地址
	Metric	3	3	6	6	Link1 的费用
Link2	ID	10.1.1.5	10.1.1.6	10.1.1.1	10.1.1.12	所连路由器的 RounterID
	IP	10.1.1.9	10.1.1.13	10.1.1.10	10.1.1.14	Link2 的本地 IP 地址
	Metic	2	4	2	4	Link2 的费用
Net1	Prefix	192.1.1.0/24	192.1.6.0/24	192.1.7.0/24	192.1.7.0/24	直连网络 Net1 的网络前缀
	Metric	1	1	1	1	到达直连网络 Net1 的费用

请回答下列问题：

(1) 本题中的网络可抽象为数据结构中的哪种逻辑结构？

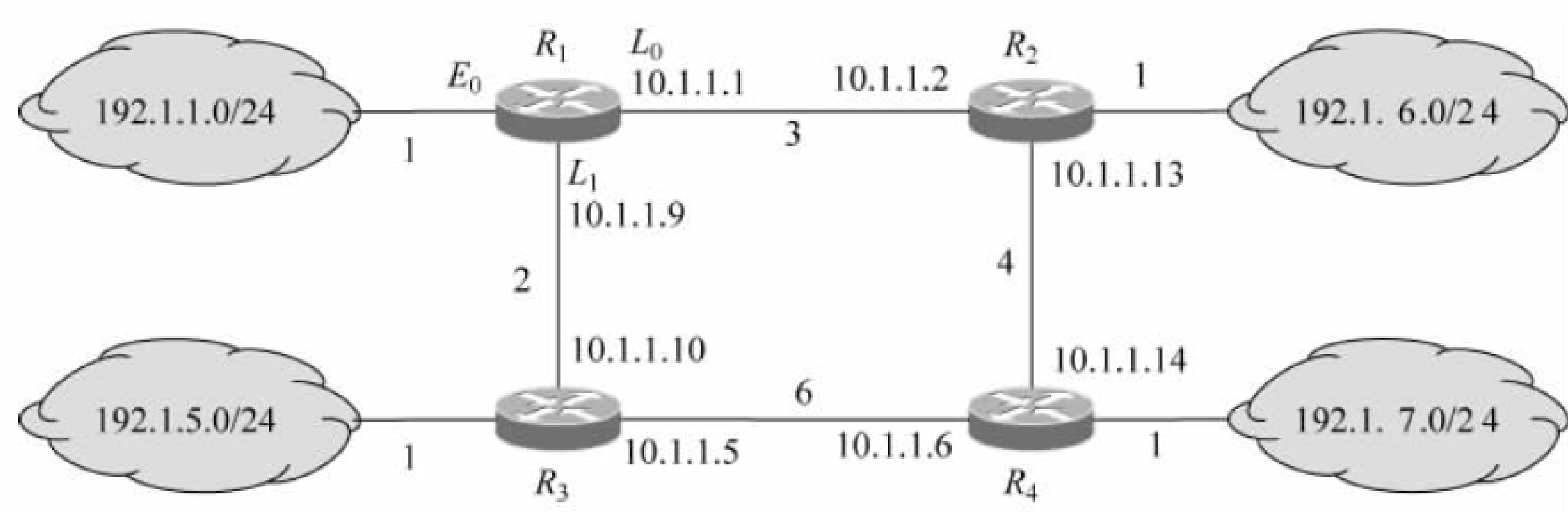


图 C.3 R_1 构造的网络拓扑

(2) 针对表 C.1 中的内容设计合理的链式存储结构，以保存该表中的链路状态信息(LSI)。要求给出链式存储结构的数据类型定义，并画出对应表 C.1 的链式存储结构示意图(示意图中可仅以 ID 标识结点)。

(3) 按照迪杰斯特拉(Dijikstra)算法的策略，依次给出 R_1 到达图 C.3 中子网 192.1.x.x 的最短路径及费用。

2014 年试题参考答案

一、单项选择题(每小题 2 分，只有一个选项是符合题目要求的)

1. C。设该程序段的执行时间为 $T(n)$，有：

$$T(n)=\sum_{k=1}^{\log_2 n}\sum_{j=1}^{n}1=n\log_2 n=O(n\log_2 n)$$

2. B。将中缀表达式 $a/b+(c*d-e*f)/g$ 转换为等价的后缀表达式的过程如表 C.2 所示。

表 C.2 中缀表达式转换为后缀表达式的过程

扫描的字符	栈(栈底⇨栈顶)	后缀表达式	说　明
a		a	
/	/	a	
b	/	$a\ b$	
+	+	$a\ b$ /	出栈'/'，将'+'进栈
(	+(	$a\ b$ /	
c	+(	$a\ b/\ c$	
*	+(*	$a\ b/\ c$	
d	+(*	$a\ b/\ c\ d$	
−	+(−	$a\ b/\ c\ d$ *	出栈' * '，将'−'进栈
e	+(−	$a\ b/\ c\ d\ *\ e$	
*	+(− *	$a\ b/\ c\ d\ *\ e$	
f	+(− *	$a\ b/\ c\ d\ *\ e\ f$	

3. A。和采用队头指针指向队头元素的前一个位置，队尾指针指向队尾元素完全相同，队空条件为队头指针==队尾指针；队满条件为(队尾指针+1) mod M==队头指针。

4. D。该二叉树的中序序列为 debxac，由于 x 是叶子结点，其左、右指针都可以线索化，分别指向前驱 b 结点和后继 a 结点。

5. C。对于 F 中的每个叶子结点 s，它没有孩子，也就没有最左边的孩子，当 F 转换为 T 时，s 的左孩子指针一定为空。

6. D。在选项 D 中，110 是 1100 的前缀码。

7. D。对于选项 A 和 B，选择 3、1 后，2 的入度并不为 0。对于选项 C，5、6 顺序是错误的。

8. D。散列函数和装填（装载）因子是事先确定的。堆积现象会直接导致平均查找长度增大。

9. D。最少关键字个数＝Min＝$\lceil m/2 \rceil-1=1$，每个结点的关键字个数为 1，此时结点个数最多。

10. B。$n=9$，若 $d=2$、4 和 5，分的各个组是无序的。若 $d=3$，分为 3 组，相距 3 个位置的元素属于一个组，(9，13，20)、(1，7，23)和(4，8，15)都是有序的。

11. C。快速排序每趟归位一个元素，第 2 趟后至少有两个元素归位。这 7 个元素的排序结果为 2，3，4，5，6，7，9，选项 C 只有一个元素归位，所以是不可能的。

二、综合应用题（共两小题，共 23 分）

1. 设 $f(b,l)$ 为以 b 为根结点（其层次为 l）的 WPL，对应的递归模型如下：

$$f(b,l)=\begin{cases} b\text{—> weight} * (l-1) & \text{当 } b \text{ 结点为叶子结点时} \\ f(b\text{—> left}, l+1)+f(b\text{—> right}, l+1) & \text{其他情况} \end{cases}$$

对应的算法如下：

```
typedef struct node
{    int weight;
     struct node * left, * right;
} BTNode;
int WPL1(BTNode * b,int level)            //求 b 结点(其层次为 level)的 WPL
{    if (b -> left == NULL && b -> right == NULL)
         return b -> weight * (level - 1);
     else
         return WPL1(b -> left, level + 1) + WPL1(b -> right, level + 1);
}
int WPL(BTNode * root)                    //求 root 的 WPL
{
     return WPL1(root,1);
}
```

2. (10 分)(1) 本题中的网络可抽象为数据结构中的图，如图 C.4 所示。

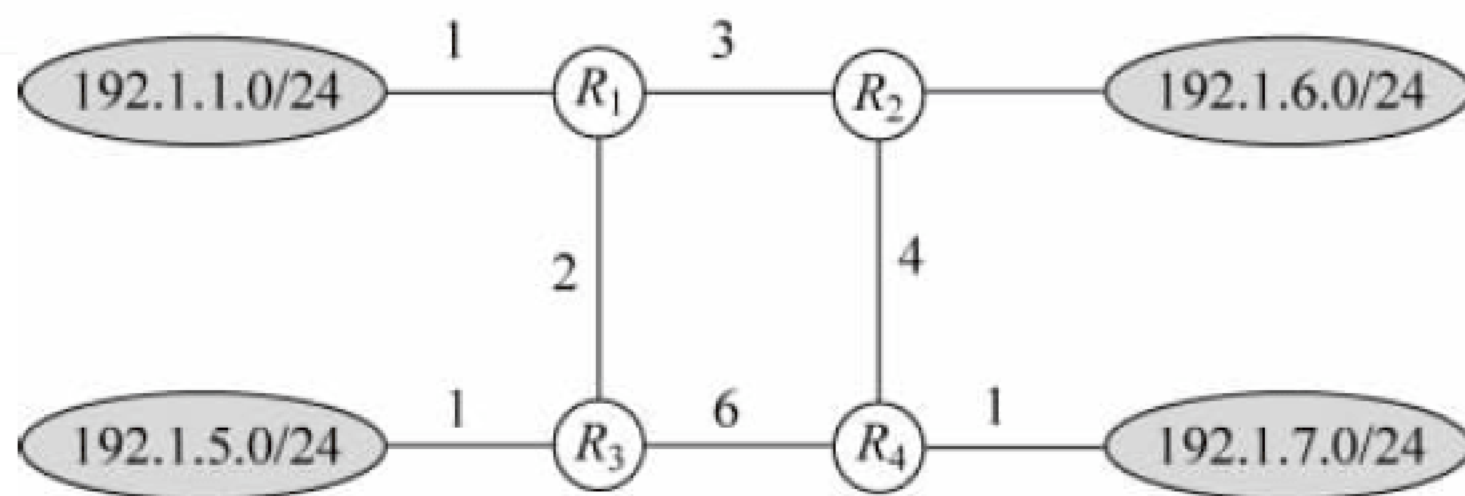


图 C.4　网络抽象的图结构

(2) 链式存储结构的数据类型定义如下:

```
typedef struct
{   unsigned int ID;
    unsigned int IP;
} LinkNode;                          //Link 结点类型
typedef struct
{   unsigned int prefix;
    unsigned int mask;
} NetNode;                           //Net 结点类型
typedef struct Node
{   int flag;                        //flag = 1:Link flag = 2:Net
    union
    {   LinkNode Lnode;
        NetNode Nnode;
    } LinkOrNet;
    unsigned int Metric;
    struct Node * next;
} ArcNode;                           //邻接表的边结点
typedef struct HNode
{   unsigned int RouterID;
    ArcNode * LN_link;
    struct HNode * next;
} HNODE;                             //邻接表的表头结点
```

表 C.1 的链式存储结构示意图如图 C.5 所示。

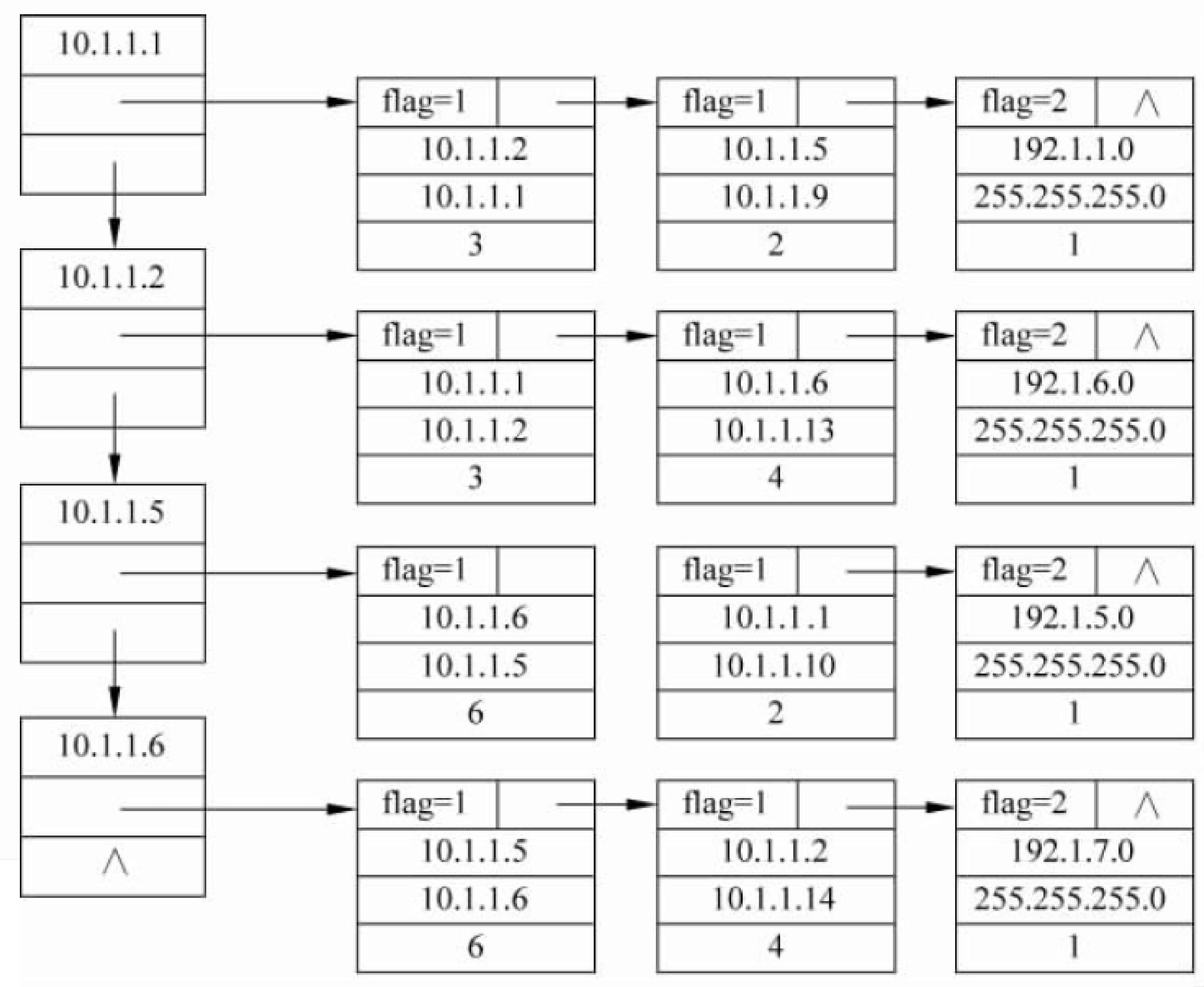


图 C.5　表 C.1 的链式存储结构示意图

(3) 按照 Dijkstra 算法的策略,计算 R_1 到达各子网 192.1.x.x 的最短路径及费用如表 C.3 所示(实际上该图十分简单,可以直接找出从 R_1 到各子网结点的最短路径及费用)。

表 C.3 计算的最短路径及费用

	目的网络	路径	代价(费用)
步骤 1	192.1.1.0/24	直接到达	1
步骤 2	192.1.5.0/24	$R_1 \to R_3 \to$ 192.1.5.0/24	3
步骤 3	192.1.6.0/24	$R_1 \to R_2 \to$ 192.1.6.0/24	4
步骤 4	192.1.7.0/24	$R_1 \to R_2 \to R_4 \to$ 192.1.7.0/24	8

2015 年试题

一、单项选择题(每小题 2 分,只有一个选项是符合题目要求的)

1. 已知程序如下:

```
intS(int n)
{     return (n<=0) ? 0 :S(n-1) + n;   }
void main()
{     cout << S(1); }
```

程序运行时使用栈来保存调用过程的信息,自栈底到栈顶保存的信息依次对应的是________。

A. main()→S(1)→S(0)　　B. S(0)→S(1)→main()

C. main()→S(0)→S(1)　　D. S(1)→S(0)→main()

2. 先序序列为 a、b、c、d 的不同二叉树的个数是________。

A. 13　　B. 14　　C. 15　　D. 16

3. 下列选项给出的是从根分别到达两个叶子结点路径上的权值序列,能属于同一棵哈夫曼树的是________。

A. 24,10,5 和 24,10,7　　B. 24,10,5 和 24,12,7

C. 24,10,10 和 24,14,11　　D. 24,10,5 和 24,14,6

4. 现在有一棵无重复关键字的平衡二叉树(AVL 树),对其进行中序遍历可得到一个降序序列。下列关于该平衡二叉树的叙述正确的是________。

A. 根结点的度一定为 2　　B. 树中最小元素一定是叶子结点

C. 最后插入的元素一定是叶子结点　　D. 树中最大元素一定是无左子树

5. 设有向图 $G=(V,E)$,顶点集 $V=\{v_0,v_1,v_2,v_3\}$,边集 $E=\{<v_0,v_1>,<v_0,v_2>,<v_0,v_3>,<v_1,v_3>\}$,若从顶点 v_0 开始对图进行深度优先遍历,则可能得到的不同遍历序列个数是________。

A. 2　　B. 3　　C. 4　　D. 5

6. 在求图 C.6 所示带权图的最小(代价)生成树时,可能是克鲁斯卡(Kruskal)算法第 2 次选中但不是普里姆(Prim)算法(从 v_4 开始)第 2 次选中的边是________。

A. (v_1,v_3)　　B. (v_1,v_4)

C. (v_2,v_3)　　D. (v_3,v_4)

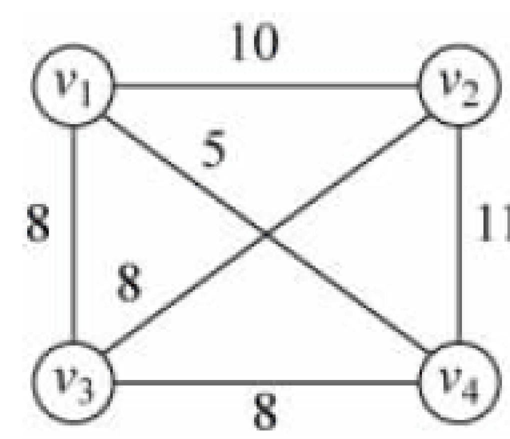


图 C.6 一个带权图

7. 在下列选项中，不能构成折半查找中关键字比较序列的是________。

A. 500,200,450,180　　B. 500,450,200,180

C. 180,500,200,450　　D. 180,200,500,450

8. 已知字符串 s 为"abaabaabacacaabaabcc"，模式串 t 为"abaabc"，采用 KMP 算法进行匹配，第一次出现"失配"($s[i]!=t[i]$)时 $i=j=5$，则下次开始匹配时 i 和 j 的值分别是________。

A. $i=1,j=0$　　B. $i=5,j=0$　　C. $i=5,j=2$　　D. $i=6,j=2$

9. 下列排序算法中元素的移动次数和关键字的初始排列次序无关的是________。

A. 直接插入排序　　B. 起泡排序　　C. 基数排序　　D. 快速排序

10. 已知小根堆为 8,15,10,21,34,16,12，删除关键字 8 之后需重建堆，在此过程中关键字之间的比较数是________。

A. 1　　B. 2　　C. 3　　D. 4

11. 希尔排序的组内排序采用的是________。

A. 直接插入排序　　B. 折半插入排序　　C. 快速排序　　D. 归并排序

二、综合应用题(共两小题，共 23 分)

1. (15 分)用单链表保存 m 个整数，结点的结构为(data,link)，且$|data|\leqslant n$(n 为正整数)。现要求设计一个时间复杂度尽可能高效的算法，对于链表中绝对值相等的结点，仅保留第一次出现的结点而删除其余绝对值相等的结点。例如，若给定的单链表 head 如图 C.7(a)所示，则删除结点后的 head 如图 C.7(b)所示。

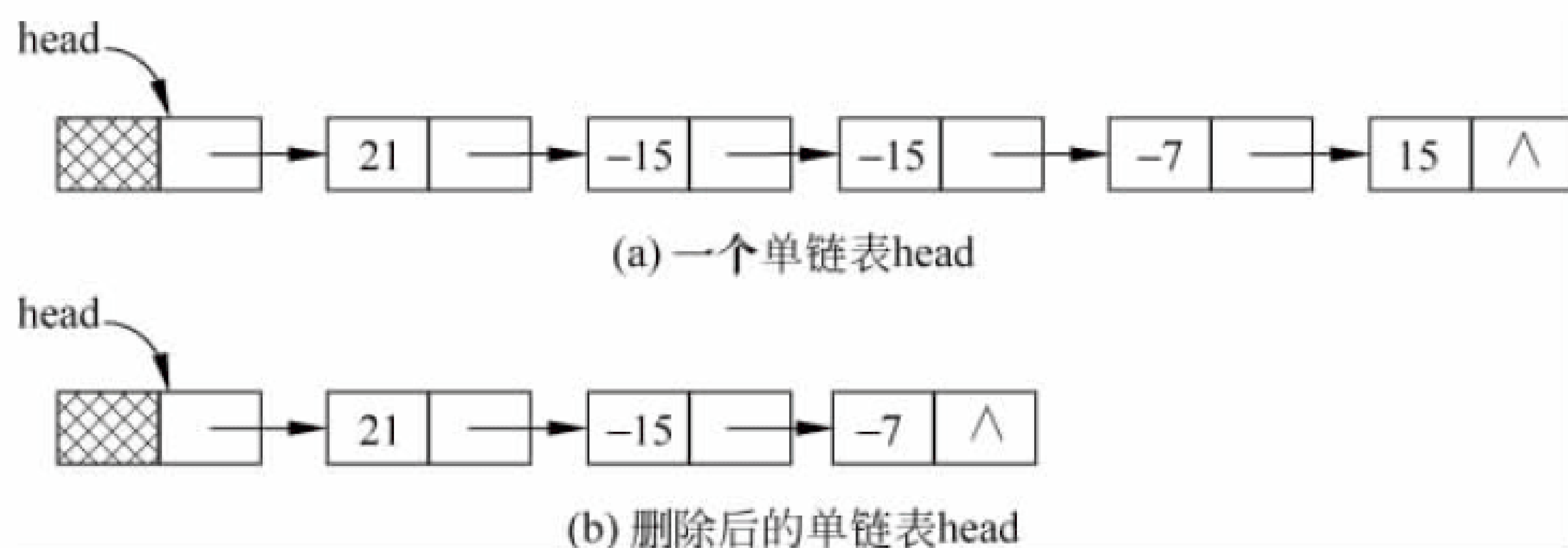


图 C.7　一个单链表及其删除后的结果

要求：

(1) 给出算法的基本思想。

(2) 使用 C 或 C++语言，给出单链表结点的数据类型定义。

(3) 根据设计思想，采用 C 或 C++语言描述算法，关键之处给出注释。

(4) 说明所涉及算法的时间复杂度和空间复杂度。

2. (8 分)已知有 5 个顶点的图 G 如图 C.8 所示。

请回答下列问题：

(1) 写出图 G 的邻接矩阵 A(行、列下标从 0 开始)。

(2) 求 $\boldsymbol{A}^2$，矩阵 $\boldsymbol{A}^2$ 中位于 0 行 3 列元素值的含义是什么？

(3) 若已知具有 $n(n\geqslant 2)$个顶点的邻接矩阵为 $\boldsymbol{B}$，则 $\boldsymbol{B}^m(2\leqslant m\leqslant n)$非零元素的含义是什么？

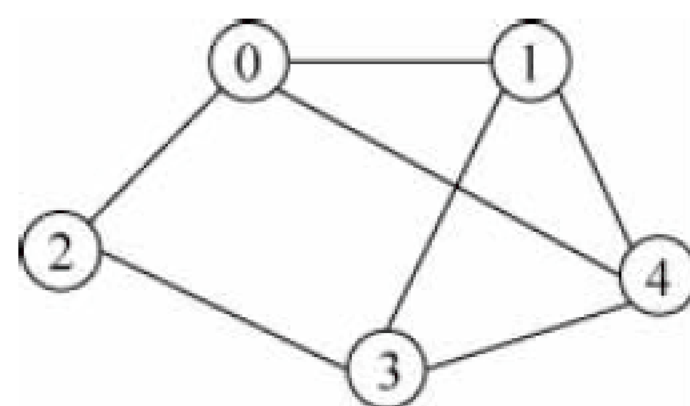


图 C.8　一个无向图

2015 年试题参考答案

一、单项选择题（每小题 2 分，只有一个选项是符合题目要求的）

1. A。首先从 main()开始执行程序，将 main()信息进栈，遇到调用 S(1)，将 S(1)信息进栈，在执行递归函数 S(1)时又遇到调用 S(0)，再将 S(0)信息进栈。所以，自栈底到栈顶保存的信息的顺序是 main()→S(1)→S(0)。

2. B。这里 $n=4$，可以构造的不同二叉树的个数$=\frac{1}{n+1}C_{2n}^{n}=14$。

3. D。对于选项 A，24 为根，两个 10 不可能是根的两个孩子。对于选项 B，24 为根，10 和 12 不可能是根的两个孩子。对于选项 C，10 不可能是 10 的孩子。

4. D。该平衡二叉树的中序遍历可得到一个降序序列，说明左子树所有结点的关键字大于根结点关键字，右子树所有结点的关键字小于根结点关键字。树中的最大元素一定是根结点的最左下结点，它没有左孩子；树中的最小元素一定是根结点的最右下结点，它没有右孩子。

5. D。对应的图如图 C. 9 所示，从顶点 v_0 出发的深度优先遍历序列有 $v_0v_1v_3v_2$、$v_0v_2v_1v_3$、$v_0v_2v_3v_1$、$v_0v_3v_1v_2$、$v_0v_3v_2v_1$，共 5 个序列。

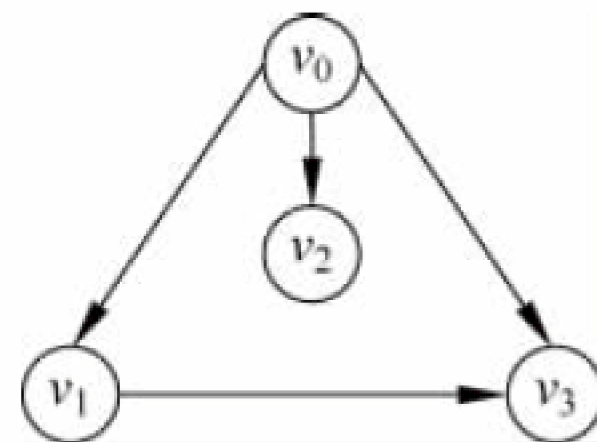


图 C. 9　一个有向图

6. C。克鲁斯卡(Kruskal)算法首先选取边(v_1,v_4)：5，剩下的最小边有 3 条，即(v_1,v_3)：8、(v_3,v_4)：8、(v_2,v_3)：8，可以任意选择一条。若采用普里姆(Prim)算法（从 v_4 开始），首先取边(v_1,v_4)：5，构成$\{v_1,v_4\}$和$\{v_2,v_3\}$两个顶点集合，下一步只能选取这两个顶点集合之间的边，不可能选择边(v_2,v_3)。

7. A。4 个选项对应的关键字比较过程如图 C. 10 所示，显然图 C. 10(a)是不可能的。实际上，折半查找中的关键字比较序列对应的树一定是一棵二叉排序树。

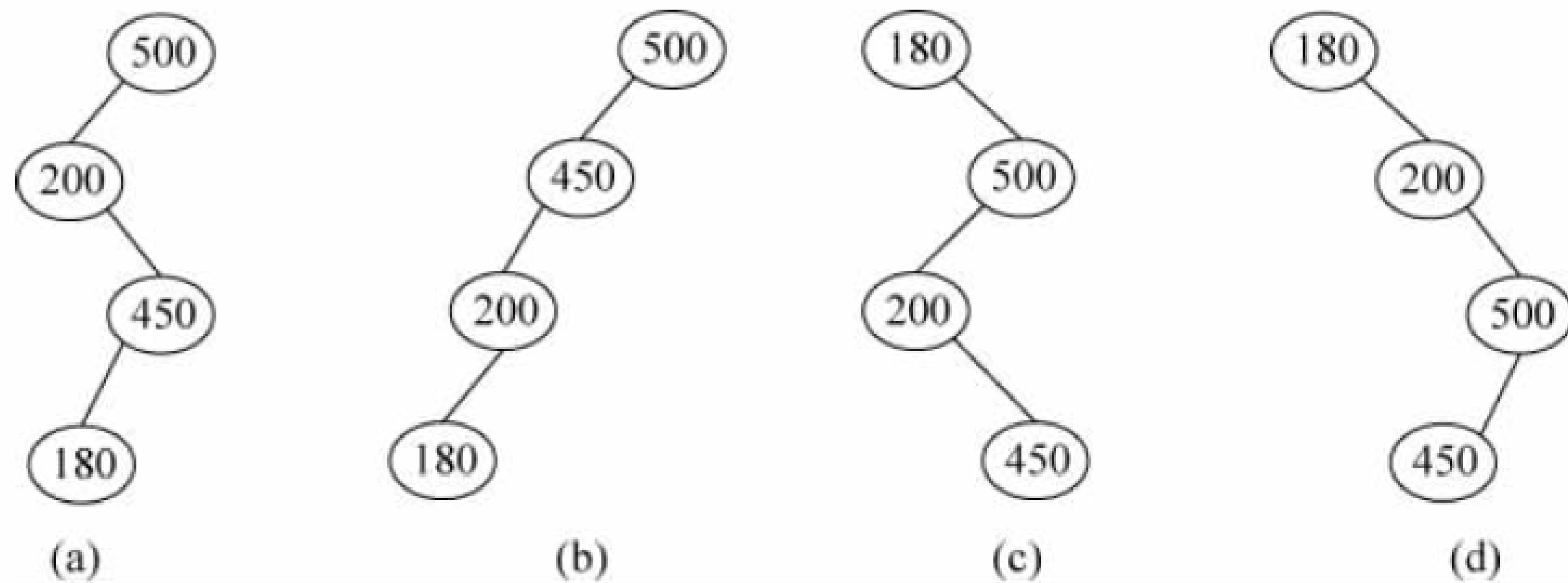


图 C. 10　4 个关键字比较过程

8. C。失配的位置为 $i=j=5$，在 KMP 算法中，i 不变，$j=\text{next}[j]$，在模式串 t 中，$t[5]=$'c'，它的前面有“ab”和开头的两个字符相同，所以 next[5]=2。

9. C。采用链表的基数排序中没有元素移动。

10. C。删除 8 筛选为堆的过程如图 C. 11 所示。在图 C. 11(b)中，设 tmp=12，根的两个孩子 15、10 比较一次，取较小者 10，再与 tmp 比较一次，10<tmp，将 10 放在根中；原来 10 结点只有一个孩子，将孩子 16 与 tmp 比较一次，16>tmp，最后将 tmp 放在原来的 10 结

点处。一共比较 3 次。

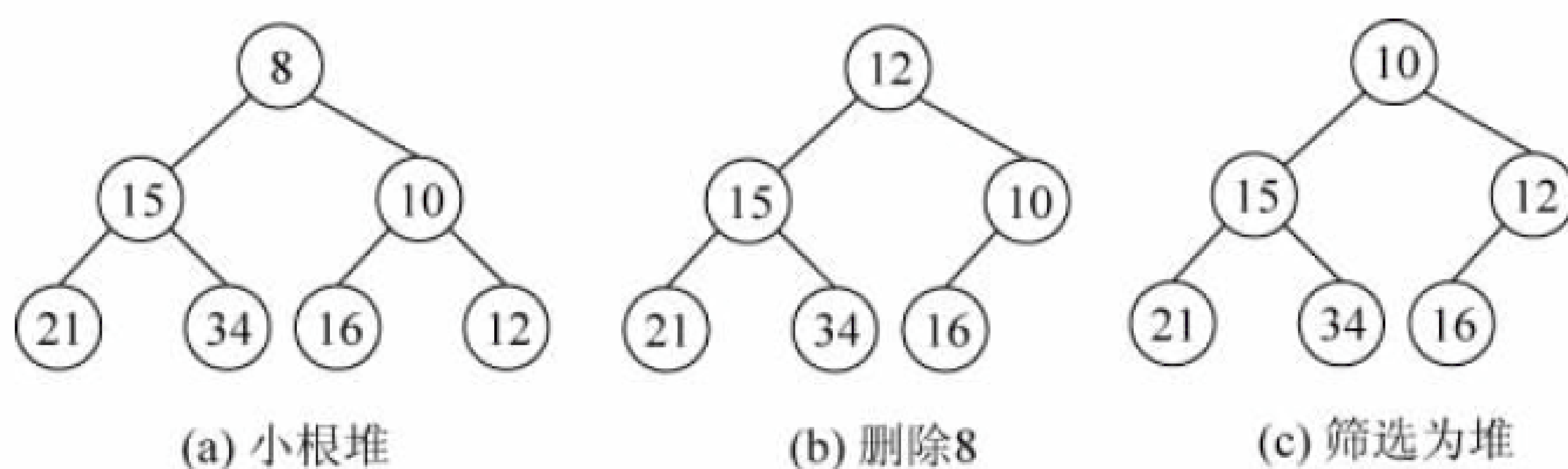


图 C.11　删除 8 筛选为堆的过程

11. A。希尔排序每一趟将所有元素分为 d 组，每组采用直接插入排序。

二、综合应用题(共两小题，共 23 分)

1. (15 分)(1) 由于|data|≤n，设置一个 $b[0..n]$的数组(大小为 $n+1$)，初始化所有元素为 0，用 p 扫描单链表 head 的结点，pre 指向其前驱结点，当 data 域为|p—> data|的结点首次出现时，必有 b[|p—> data|]=0，保留结点 p，让 b[|p—> data|]增 1；若 b[|p—> data|]不为 0，则结点 p 是重复结点，通过 pre 结点删除它。

(2) 单链表的结点类型声明如下：

```
typedef struct node
{    int data;
     struct node * next;
} LinkNode;
```

(3) 对应的算法如下：

```
void fun(LinkNode *&head, int n)
{    LinkNode * pre = head, * p = head -> next;
     int * b,m;
     b = (int * )malloc(sizeof(sizeof(int) * (n + 1));
     for (int i = 0;i < n + 1;i++)              //b 数组的所有元素初始化为 0
         b[i] = 0;
     while (p!= NULL)
     {    m = p -> data > 0?p -> data: - p -> data;
          if (b[m] == 0)                        //结点值为|p -> data|首次出现
          {    b[m]++;
               pre = p;                         //pre、p 同步后移
               p = p -> next;
          }
          else
          {    pre -> next = p -> next;         //删除结点 p
               free(p);
               p = pre -> next;                 //p 指向 pre 结点的后继结点
          }
     }
     free(b);                                   //释放 b 的存储空间
}
```

(4) 该算法只遍历一次链表,所以时间复杂度为 $O(m)$,m 为单链表 head 中的数据结点个数。该算法申请大小为 n 的数组,所以空间复杂度为 $O(n)$,n 为结点绝对值的最大值。

2. (8 分)(1) 图 G 的邻接矩阵 $\boldsymbol{A}$ 如下:

$$\begin{bmatrix} 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 \end{bmatrix}$$

(2) $\boldsymbol{A}^2$ 如下:

$$\begin{bmatrix} 3 & 1 & 0 & 3 & 1 \\ 1 & 3 & 2 & 1 & 2 \\ 0 & 2 & 2 & 0 & 2 \\ 3 & 1 & 0 & 3 & 1 \\ 1 & 2 & 2 & 1 & 3 \end{bmatrix}$$

其中,0 行 3 列的元素为 3,表示顶点 0 到顶点 3 的长度为 2 的路径共有 3 条。

(3) $\boldsymbol{B}^m(2\leqslant m\leqslant n)$中位于 i 行 j 列的非零元素的含义是图中从顶点 i 到顶点 j 的长度为 m 的路径条数。

图书资源支持

感谢您一直以来对清华版图书的支持和爱护。为了配合本书的使用，本书提供配套的素材，有需求的用户请到清华大学出版社主页（http://www.tup.com.cn）上查询和下载，也可以拨打电话或发送电子邮件咨询。

如果您在使用本书的过程中遇到了什么问题，或者有相关图书出版计划，也请您发邮件告诉我们，以便我们更好地为您服务。

我们的联系方式：

地　　址：北京海淀区双清路学研大厦 A 座 707

邮　　编：100084

电　　话：010－62770175－4604

资源下载：http://www.tup.com.cn

电子邮件：weijj@tup.tsinghua.edu.cn

QQ：883604（请写明您的单位和姓名）



扫一扫

资源下载、样书申请

新书推荐、技术交流

用微信扫一扫右边的二维码，即可关注清华大学出版社公众号“书圈”。